Declarative expression and optimization of data-intensive flows

Georgia Kougka and Anastasios Gounaris

Department of Informatics
Aristotle University of Thessaloniki, Greece
{georkoug,gounaria}@csd.auth.gr

Abstract. Data-intensive analytic flows, such as populating a datawarehouse or analyzing a click stream at runtime, are very common in modern business intelligence scenarios. Current state-of-the-art data flow management techniques rely on the users to specify the flow structure without performing automated optimization of that structure. In this work, we introduce a declarative way to specify flows, which is based on annotated descriptions of the output schema of each flow activity. We show that our approach is adequate to capture both a wide-range of arbitrary data transformations, which cannot be supported by traditional relational operators, and the precedence constraints between the various stages in the flow. Moreover, we show that we can express the flows as annotated queries and thus apply precedence-aware query optimization algorithms. We propose an approach to optimizing linear conceptual data flows by producing a parallel execution plan and our evaluation results show that we can speedup the flow execution by up to an order of magnitude compared to existing techniques.

1 Introduction

Data-intensive analytic flows are typically encountered in business intelligence scenarios and are nowadays attracting renewed interest, since they go beyond traditional Extract - Transform - Load (ETL) flows [19, 16]. ETLs are a special form of data flows used to populate a data warehouse with up-to-date, clean and appropriately transformed source records. They can be considered as a directed acyclic graph (DAG), similar to scientific and business workflows, capturing the flow of data from the sources to the data warehouse [18]. Next generation business intelligence (BI) involves more complex flows that encompass data/text analytics, machine learning operations, and so on [16]; in this work we target such BI flows.

Our motivation is twofold. Firstly, modern data flows may be particularly complex to be described manually in a procedural manner (e.g., [16]). Secondly, the vast amount of data that such flows need to process under pressing time constraints calls for effective, automated optimizers, which should be capable of devising execution plans with minimum time cost. In this work, we target two correlated goals, namely declarative statement and efficient optimization.

Declarative statement of data flows implies that, instead of specifying the exact task ordering, flow designers may need to specify only higher-level aspects, such as the precedence constraints between flow stages, i.e., which task needs to precede other tasks. An example of an existing declarative approach is the Declare language that is based on linear temporal logic [12]. We follow a different approach that bears similarities with data integration mediation systems and allows the flow to be expressed in the form of annotated SQL-like queries.

Regardless of the exact declarative form a flow can be expressed, such declarative approaches are practical only under the condition that the system is capable of taking the responsibility for automatically devising a concrete execution plan in an efficient and dependable manner; this is exactly the role of query optimization in database systems, which also rely on declarative task specifications, and we envisage a similar role in data flow systems as well. Although traditional query optimization techniques cannot be applied in a straightforward manner, we propose an approach to optimizing linear conceptual data flows by producing a parallel execution plan, inspired by advanced query optimization techniques.

The contribution of this work is as follows. We demonstrate how we can express data flows in a declarative manner that is then amenable to optimization in a straight-forward manner. To this end, we use an annotated flavour of SQL, where flow steps are described by input and output virtual relations and annotations are inspired by the binding patterns in [5]. Our approach to declarative statement does not rely on the arguably limited expressiveness of relational algebra in order to describe arbitrary data manipulations, like those in ETLs, and is adequate to describe the precedence constraints between data flow tasks. In addition, we present optimization algorithms for logically linear flows that take into account the precedence constraints so that correctness is guaranteed. As shown in our evaluation, the approach that allows for parallel execution flows may lead to performance improvements up to an order of magnitude in the best case, and performance degradation up to 1.66 times in the worst case compared to the best current technique.

Structure: Sec. 2 presents our approach to declarative statement of data flows with a view to enabling automated optimization. The optimization algorithms for linear flows along with thorough evaluation of performance improvements are in Sec. 3. In Sec. 4 and 5 we discuss related work and conclusions, respectively.

2 Declarative statement of data flows

We map each flow activity[1] to a virtual relation described by a non-changing schema. More specifically, each activity is mapped to a virtual annotated relation $R(A, a, p)$, where (i) R is the task's unique name that also serves as its identifier; (ii) $A = (A_1, A_2, \cdots, A_n)$ is the list of input and output attributes, which are also identified by their names; (iii) a is a vector of size equal to the size of A, such that the i-th element of a is "b" (resp. "f") if the i-th element of A must be *bound* (resp. *free*); and (iv) p is a list of sets, where the j-th set includes the names of the *bound* variables of other virtual relations that must precede $R.A_j$.

[1] The terms flow tasks and activities are used interchangeably.

The notation of the a vector is aligned to the notation of *binding patterns* in [5], and allows us to distinguish between the attributes that need to belong to the input (the *bound* ones) and the new attributes that are produced in the output (*free* attributes). In other words, a binding pattern for a relation R means that the attributes of R annotated with b must be given as inputs when accessing the tuples of R, whereas the attributes annotated by f denote the new attributes derived by the task invocation. For example, the relation $Task1(A : (X, Y, Z),\ a : (bbf),\ p : (\{Task2.X\}\{NULL\}\{NULL\}))$ corresponds to an activity called *Task1*, which needs to be given the values of the X and Y attributes as input and returns a new attribute Z. Attribute X must first be processed by $Task2$. For brevity, this relation can also be written as $Task1(X^{b,Task2}, Y^b, Z^f)$ Additionally, we treat data sources as specific data-producing activities, where all attributes are annotated with f. Linear conceptual flows comprise a single data source.

The following statements hold: (a) The output data items of each flow task are regarded as tuples, the schema of which conforms to the virtual relations introduced above. (b) Data sources are treated as specific data-producing activities, where all attributes are annotated with f. (c) The flow tasks, even when they can be described by standard relational operators (e.g., when they simply filter data), they are always described as virtual relations. (d) The relations can be combined with standard relation operators, such as joins and unions; concrete examples are given in the sequel. (e) For each attribute X that is *bound* in relation R, there exists a relation R', which contains attribute X with a *free* annotation. (f) A task outputing a *free* attribute must precede the tasks that employ the same attributes as *bound* attributes in their schema. (g) Simply relying on b/f annotations is inadequate for capturing all the precedence constraints in ETL workflows, where there may exist a *bound* attribute that is manipulated by a filtering task and also appears in the *bound* grouping values of an aggregate function: in that case, the semantics of the flow may change if we swap the two activities, as also shown in [6]. For that reason, it is always necessary to define the p list of each activity. (h) Although most ETL transformation can be described by static schemas, there may be data flow activities, such as some forms of pivots/unpivots [3] that cannot be mapped to the virtual relations as defined above, because the schema of their output cannot be always defined *a-priori*. (i) Tasks need not correspond to ETL transformation solely; they can also encompass intermediate result storage.

Precedence constraints of a flow form a directed acyclic graph (DAG) G in which there is a node corresponding to each flow task and directed edges from one task to another define the presence of precedence constraint between them. A main goal of the annotations is to fully capture the precedence constraints among tasks. This goal is attained because the edges in the precedence graph can be derived from the p list of each activity and the (f) item above.

2.1 Flow Examples

Linear Flows Our first example is taken from [18] and is illustrated in Fig. 1. It is a linear flow that applies a set of filters, transformations, and aggregations to a

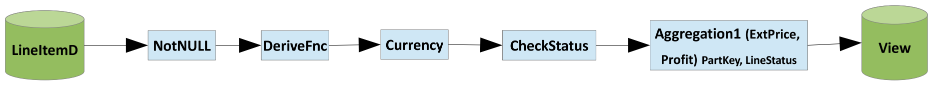


Fig. 1. A linear ETL flow.

single table from the TPC-H decision support benchmark. In particular, the flow consists of 5 activities: (i) *NotNull*, which checks the fields *PartKey, OrderKey* and *SuppKey* for NULL values. Any NULL values are replaced by appropriate special values (or the tuple is dropped). (ii) *DeriveFnc*, which calculates a value for the new field *Profit* that is derived from other fields and more specifically by subtracting the values of fields *Tax* and *Discount* from the value in *ExtPrice*. (iii) *Currency*, which alters the fields *ExtPrice, Tax, Discount* and *Profit* to a different currency. (iv) *CheckStat*, which is a filter that keeps only records whose return status is `false`. (v) *Aggregation1*, which calculates the sum of *ExtPrice* and *Profit* fields grouped by values in *PartKey* and *LineStatus*.

All activities can be mapped to virtual relations, and the whole ETL can be modelled as a Select-Project-Join (SPJ) query in order to provide online updates to the view in Fig. 1. It is important to note that the relevant attributes of the source relation, *LineItem*, are annotated as *free* attributes. Also, the $PartKey^{b,CheckStat}$ attribute in the aggregation activity contains a task annotation, which allows us to define that the aggregation must only be performed after the *CheckStat* activity in order to ensure semantic equivalence with the flow in Fig. 1. Finally, in *Aggregation1*, the attributes *PartKey* and *LineStatus* have the same values with *PartKeyGroup* and *LineStatusGroup*, respectively, but the latter are annotated as *free* attributes in order to facilitate manipulation statements that build on the grouped values.

```
Select  PartKeyGroup, LineStatusGroup, UpdatedSumProfit,
    UpdatedSumExtPrice
From    LineItem(PartKey^f,OrderKey^f,SuppKey^f,Discount^f,Tax^f,ExtPrice^f,···) ⋈
        NotNull(PartKey^b,OrderKey^b,SuppKey^b) ⋈
        DeriveFnc(PartKey^b,Profit^f) ⋈
        Currency(PartKey^b,ExtPrice^b,Discount^b,Profit^b,Tax^b) ⋈
        CheckStat(PartKey^b,ReturnStatus^b ⋈
        Aggregation1(PartKey^{b,CheckStat}, LineStatus^b, Profit^b, ExtPrice^b,
                     ReturnStatus^b,LineStatusGroup^f,PartKeyGroup^f,
                     UpdatedSumProfit^f,UpdatedSumExtPrice^f)
Where    LineItem.PartKey = NotNull.PartKey and
         LineItem.PartKey = DeriveFnc.PartKey and
         LineItem.PartKey = Currency.PartKey and
         LineItem.PartKey = CheckStat.PartKey and
         CheckStat.PartKey = Aggregation1.PartKey
```

In the above example there are several precedence constraints that can automatically derived from the annotated query: *LineItem* must precede all other activities, *DeriveFnc* must precede *Currency* and *Aggregation1*, whereas *CheckStat* must precede *Agregation1* as well. Although those constraints seem restrictive, they do not preclude other flow structures, e.g., *CheckStat* to be applied earlier and *Currency* to be applied at the very end to decrease the number of total currency transformations.

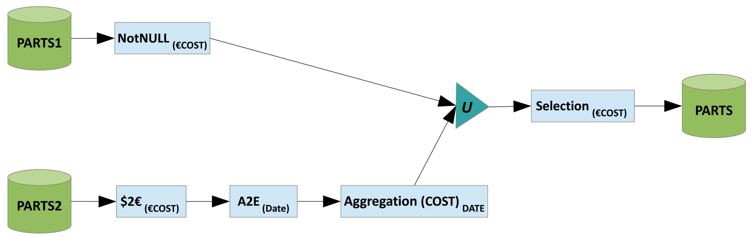


Fig. 2. A more complex ETL flow.

More complex flows Fig. 2 shows a more complex flow on top of two real data sources, also taken from [18]. The tasks employed are: (i) *NotNull*, which checks the field *Cost* for NULL values, so that such values are replaced or the tuple is dropped. (ii) *dollar2euro*, which changes the values in *Cost* from dollars to euros. (iii) *A2E*, which alters the format of the field *Date* from american to european. (iv) *Aggregation*, which calculates the sum of costs grouped by date, (v) *Selection*, which filters the (aggregated) cost field according to a user-defined threshold. Although we can describe this flow as a complex nested query, for clarity, we use two SPJ sub-queries. Note that, if the flow contains branches, these can be modeled as separate sub-queries in a similar manner. Also, although the selection task can easily be described by a simple select relational operator, we treat it as a separate relation.

```
Query I:
WITH Q ($PKEY$, $COST$, $DATE$) AS (
(Select *
 From   PARTS1($PKEY^f$, $COST^f$, $DATE^f$) ⋈
        NotNULL($PKEY^b$, $COST^b$)
 Where  PARTS1.PKEY = NN.PKEY )
UNION
(Select $PKEY$, $UpdatedAggCOST$, $DATEgroup$
 From   PARTS2($PKEY^f$,$COST^f$,$DATE^f$) ⋈
        dollar2euro($PKEY^b$,$COST^b$) ⋈
        A2E($PKEY^b$,$DATE^b$) ⋈
        Aggregation($PKEY^b$,$DATE^b$,$COST^b$,$DATEgroup^f$,$UpdatedAggCOST^f$)
 Where  PARTS2.PKEY = dollar2euro.PKEY and
        PARTS2.PKEY = A2E.PKEY and
        PARTS2.PKEY = Aggregation.PKEY )
)
Query II:
(Select *
 From   Q ($PKEY^f$, $COST^f$, $DATE^f$) ⋈
        Selection ($PKEY^b$, $COST^b$)
 Where  Q.PKEY = Selection.PKEY )
```

Real-world analytic flow The data flow, which is depicted in Fig. 3, shows a real-world, analytic flow that combines streaming free-form text data with structured, historical data to populate a dynamic report on a dashboard [16]. The report combines sales data for a product marketing campaign with sentiments about that product gleaned from tweets crawled from the Web and lists total sales and average sentiment for each day of the campaign. There is a single streaming source that outputs tweets on products and the flow accesses four other static sources through lookup operations.

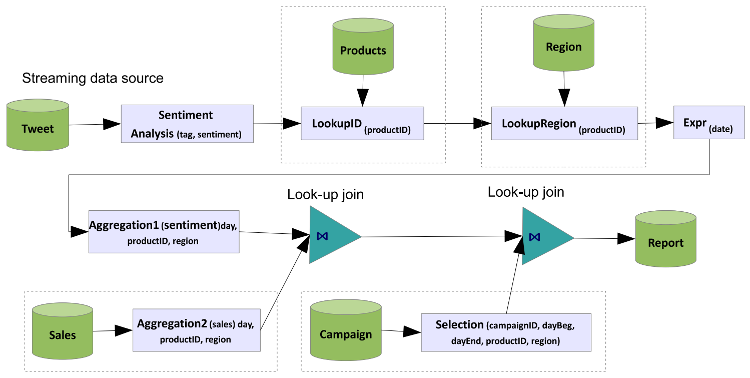

Streaming data source
Tweet
Sentiment Analysis (tag, sentiment)
Products
LookupID (productID)
Region
LookupRegion (productID)
Expr (date)
Aggregation1 (sentiment)day, productID, region
Look-up join
⋈
Look-up join
⋈
Report
Sales
Aggregation2 (sales) day, productID, region
Campaign
Selection (campaignID, dayBeg, dayEnd, productID, region)

Fig. 3. A real-world analytic flow.

The exact flow is as follows. When a tweet arrives as a timestamped string attribute (*tag*), the first task is to compute a single sentiment value in the range [-5 +5] for the product mentioned in the tweet. Then, two lookup operations are performed: the former maps product references in the tweet and the later maps geographic information (latitude and longitude) in the tweet to a geographical region (*region* attribute in the figure). The *Expr* task converts the tweet timestamp to a date. Then, the sentiment values are averaged over each region, product, and date. On a parallel path, the sales data have been rolled up to produce total sales of each product for each region and day. The rollups for sales and sentiment are joined in a pipelined fashion and finally the specific campaign of interest is selected and used to filter the result based on the information of the campaign data store [16]. In this final stage, we consider that the *Sales* and *Campaign* non-streaming sources are hidden behind the *Aggregation2* and *Selection* look-up tasks, respectively. The annotated query that describes this flow is shown below.

```
Select *
From   Tweet(tag^f,timestamp^f) ⋈
       Sentiment_Anal(tag^b,sentiment^f) ⋈
       LookupID(tag^b, productID^f) ⋈
       LookupRegion(tag^b, region^f) ⋈
       Expr(tag^b,timestamp^b,date^f) ⋈
       Aggregation1(tag^b,sentiment^b,productID^b,region^b,date^b,
                    productIdGroup^f,dateGroup^f,regionGroup^f,AvgAggSentiment^f) ⋈
       Aggregation2(productID^f,region^f,date^f,totalAggSales^f) ⋈
       Selection(productID^f,campaignID^f,dayBeg^f,dayEnd^f,region^b)
Where  Tweet.tag = Sentiment_Anal.tag and
       Tweet.tag = LookupID.tag and
       Tweet.tag = LookupRegion.tag and
       Tweet.tag = Expr.tag and
       Tweet.tag = Aggregation1.tag and
       Aggregation1.productID = Aggregation2.productID and
       Aggregation1.region = Aggregation2.region and
       Aggregation1.date = Aggregation2.date and
       Aggregation1.productID = Selection.productID and
       Aggregation1.region = Selection.region
```

2.2 Are data flows queries?

The consensus up to now is that ETL and more generic data flows cannot be expressed as (multi-) queries, due to facts such as the presence of arbitrary manipulation functions that cannot be described by relational operators, and the presence of precedence constraints [4, 13]. We agree that data flows cannot be described as standard SQL queries just by regarding manipulation functions as black box user-defined functions (UDFs). Nevertheless, as shown above, we can express data flows in an *SQL-like* manner, where the distinctive features are that (i) data manipulation steps are described through virtual relations instead of relational operators or UDFs on top of real relations; and (ii) the attributes are annotated so that precedence constraints can be derived. Our methodology thus does not suffer from the limitations when mapping a flow to a complex query with as many relations as the original data sources, which loses the information about precedence constraints.

3 Optimization of linear flows

Having transformed the flow specification to an annotated query, we can treat the flows as multi-source precedence-aware queries and benefit from any existing optimization algorithms tailored to such settings. We treat flow tasks as black-box operators. Note that we do not have to use multi-way joins regardless of the numerous joins appearing in the SQL-like statements, as in [17]. In this section, we firstly define the cost model, we then propose four optimization algorithms for minimizing the total execution cost in time units, and finally, we investigate the performance benefits.

Our optimization algorithms require that each flow activity is described by the following metadata:

- *Cost* (c_i): We use $c_i = 1/r_i$ to compute response time effectively, where r_i is the maximum rate at which results of invocations can be obtained from the i-th task.
- *Selectivity* (sel_i): it denotes the average number of returned tuples per input tuple for the i-th service. For filtering operators selectivity is always below 1, for data sources and operators that just manipulate the input, it is exactly one, whereas, for operators that may produce more output records for each input record, the selectivity is above 1.
- *Input* (I_i): The size of the input of the i-th task in number of tuples per input data tuple. It depends on the product of the selectivities of the preceding tasks in the execution plan.

Our aim is to minimize the sum of the execution time of each task. As such, the optimal plan minimizes the following formula: $(I_i c_i + I_2 c_2 + ... + I_n c_n)$.

In the following, we present our optimization approaches. Due to lack of space, we present only the main rationale.

PGreedy: The rationale of the *PGreedy* optimization algorithm is to order the flow tasks in such a way that the amount of data that is received by expensive

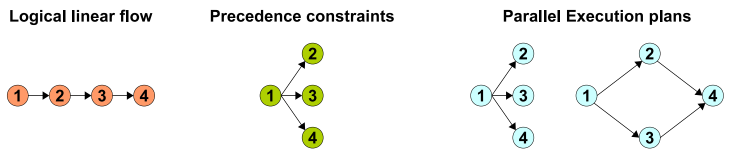


Fig. 4. A single linear conceptual data flow (left), along with its precedence constraints (middle) and two logically equivalent parallel execution plans (right).

tasks is reduced because of preceding filtering activities that prune the input dataset. Its main distinctive feature is that it allows for parallel execution plans, as shown in Fig. 4, where on the left part of the picture, a linear flow and its precedence constraints are depicted, while on the right two equivalent parallel execution plans of the same flow are presented (which both preserve all the precedence constraints). More specifically, depending on the selectivity values, the optimal execution plan may dispatch the output of an activity to multiple other activities in parallel, or place them in a sequence. To this end we adapt the algorithm in [17] with the difference that instead of considering the cost I_ic_i in each step, we consider the $(1 - sel_i)(I_ic_i)$. The latter takes into account the selectivity of the next service to be appended in the execution plan and not only the selectivity of the preceding services. We refer the reader to [17] for the rest of the details. The complexity is $O(n^5)$ in the worst case.

Swap: The *Swap* algorithm compares the cost of the existing execution plan against the cost of the transformed plan, if we swap two adjacent tasks provided that the constraints are always satisfied. We perform this check for every pair of adjacent tasks. *Swap* is proposed in [15], where, to the best of our knowledge, the most advanced algorithm for optimizing the structure of data flows is proposed. The complexity of the *Swap* algorithm is $O(n^2)$.

Greedy: Greedy algorithm is based on a typical greedy approach by adding the activity with the maximum value of $(1 - sel_i)(I_ic_i)$, which meets the precedence constraints. The time complexity of *Greedy* algorithm is $O(n^2)$. It bears similarities with the *Chain* algorithm in [21]; latter appends the activity that minimizes I_ic_i. Similarly to *Swap* and contrary to *PGreedy*, it builds only linear execution plans.

Partition: The *Partition* optimization algorithm forms clusters with activities by taking into consideration their availability. Specifically, each cluster consists from activities that their prerequisites have been considered in previous clusters. After building the clusters, each cluster is optimized separately by checking each permutation of cluster tasks. Like *Greedy*, it was first proposed for data integration systems, and the details are given in [21]. *Partition* runs in $O(n!)$ time in the worst case, and is inapplicable if a local cluster contains more than a dozen of tasks.

3.1 Experiments

In our experiments, we compare the performance of the afore-mentioned algorithms. The data flows considered consist of $n = 5, 10, 25, 50, 100$ activities and we experiment with 6 combinations of 3 selectivity value ranges and 2 sets of

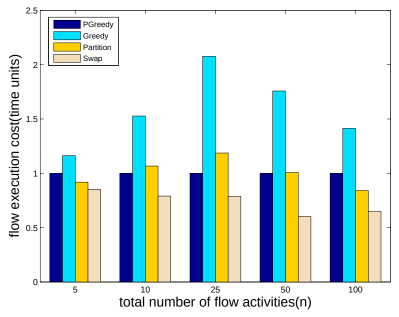


Fig. 5. Performance when *sel* $\in [0, 2]$, *cost* $\in [1, 10]$ and 25% prec. constraints.

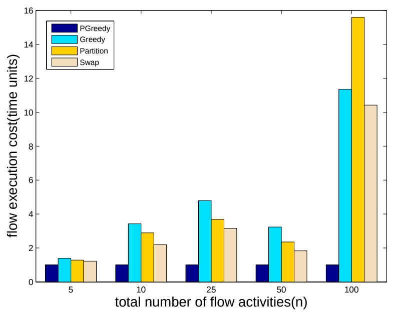


Fig. 6. Performance when *sel* $\in [0.5, 2.5]$, *cost* $\in [1, 10]$ and 25% prec. constraints.

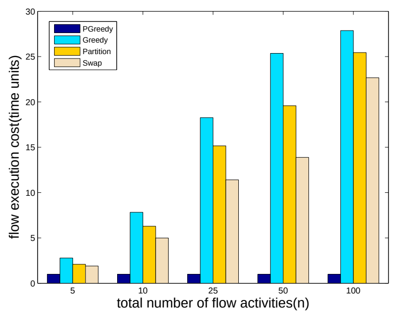


Fig. 7. Performance when *sel* $\in [1, 3]$, *cost* $\in [1, 10]$ and 25% prec. constraints.

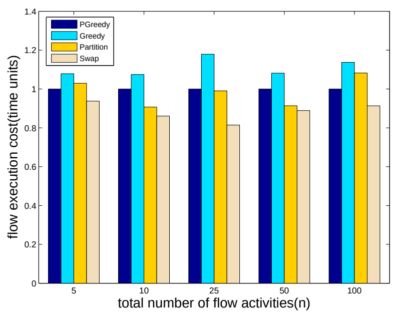
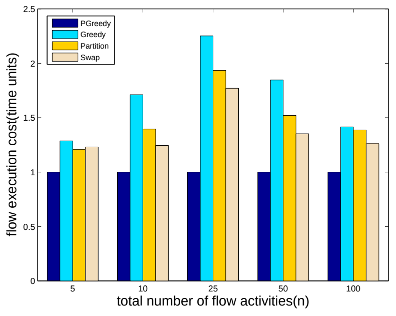


Fig. 8. Performance when *sel* $\in [0, 2]$, *cost* $\in [1, 10]$ and 50% prec. constraints.

constraint probabilities. The cost value range is the same for all the sets of experiments: $c_i \in [1, 10]$. The results correspond to the average of the data flow response time in 20 runs after removing the lowest and highest values to reduce the standard deviation. In each run, the exact selectivity, cost values and the constraints for each task are randomly generated.

In the first experiment, the selectivity values in each run are randomly generated so that *sel* $\in [0, 2]$ (thus only half of the tasks are selective) and *cost* $\in [1, 10]$ with 25% probability of having precedence constraints between two activities. The normalized results are shown in Fig. 5. A general observation in all our experiments is that *Swap* consistently outperforms *Greedy* and *Partition*. From Fig. 5, we can observe that *Swap* outperforms *PGreedy* as well. For $n = 50$, Swap is 1.66 times faster. However, as less activities are selective, *PGreedy* yields significantly lower cost than *Swap*. As shown in Figs. 6 and 7, those performance improvement may be up to 22 times (one order of magnitude).

In the following experiment, we increase the probability of having a precedence constraint between two activities. The more the constraints, the narrower the space for optimizations. The results are presented in Figs. 8-10, which follow the same pattern as above. In the worst case, *Swap* is 1.23 times faster than *PGreedy*, and, in the best case, *PGreedy* is 3.15 times faster. The general con-



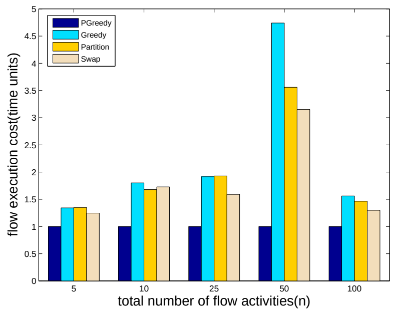


Fig. 9. Performance when $sel \in [0.5, 2.5]$, $cost \in [1, 10]$ and 50% prec. constraints.

Fig. 10. Performance when $sel \in [1, 3]$, $cost \in [1, 10]$ and 50% prec. constraints.

clusions drawn is that *Greedy* and *Partition* are never the optimal choices, and *PGreedy* outperforms *Swap* if less than half of the tasks are selective.

Regarding the time needed for the optimizations, even when $n = 100$, the time for running *PGreedy* and *Partition* is approximately a couple of seconds using a machine with an Intel Core i5 660 CPU with 6GB of RAM. Thus it can be safely considered as negligible.

4 Related Work

Modern ETL and flow analysis tools, such as Pentaho's platform[2], do not support declarative statement of flows and automated optimizations of their structure. Declare is an example of a declarative flow language [12]; contrary to our proposal, it is based on linear temporal logic and can be used only through a graphical interface in the context of Yawl[3]. Declare can capture precedence constraints, and, as such, may stand to benefit from the optimizations proposed in this work, but does not perform any optimizations in its own right.

The potential of data management solutions to enhancing the state-of-the art in workflow management has been identified since mid 2000s. An example of strong advocates of the deeper integration and coupling of databases and workflow management systems has appeared in [14]. Earlier examples of developing data-centric techniques of manipulating workflows include the prototypes described in [7, 11, 9], which allow workflow tasks to be expressed on top of virtual data tables in a declarative manner but do not deal with optimization, although they can be deemed as enabling it. Other declarative approaches to specifying workflows, such as [1, 22], are not coupled with approaches to capturing precedence constraints and optimizing the flow structure either.

Data management techniques have been explored in the context of ETL workflows for data warehouses in several proposals, e.g., [4, 13, 15]. In [15], the authors consider ETL workflows as states and use transitions to generate new states in order to navigate through the state space. The main similarity with our work is the mapping of workflow activities to schemata, which, however,

[2] `http://www.pentaho.com/`

[3] `http://www.yawlfoundation.org/`

are not annotated and thus inadequate to describe precedence constraints on their own. Focusing on the physical implementation of ETL flows, the work in [18] exploits the logical-level description combined with appropriate cost models, and introduces sorters in the execution plans. In [16], a multi-objective optimizer that allows data flows spanning execution engines is discussed.

Another proposal for flow structure optimization has appeared in [20], which decreases the number of invocations to the underlying databases through task merging. In [10], a data oriented method for workflow optimization is proposed that is based on leveraging accesses to a shared database. In [6], the optimizations are based on the analysis of the properties of user-defined functions that implement the data processing logic. Several optimizations in workflows are also discussed in [2]. Our optimization approach shown in Section 3 is different from those proposals in that it is capable of performing arbitrary correct task reordering. In our previous work, we employ query optimization techniques to perform workflow structure reformations, such as reordering or introducing new services in scientific workflows [8].

5 Conclusions

As data flows become more complex and come with requirements to deliver results under pressing time constraints, there is an increasing need for more efficient management of such flows. In this work, we focused on data-intensive analytic flows that are typically encountered in business intelligence scenarios. To alleviate the burden to manually design complex flows, we introduced a declarative way to specify such flows at a conceptual level using annotated queries. A main benefit from this approach is that the flows become amenable to sophisticated optimization algorithms that can take over the responsibility for optimizing the structure of the data flow while taking into account any precedence constraints between flow activities. We discuss optimization of linear conceptual data flows, and our evaluation results show that we can speedup the flow execution by up to an order of magnitude if we consider parallel execution plans. Our future work includes the deeper investigation of optimization algorithms to non-linear conceptual flows and the coupling of optimization techniques that reorder tasks with resource scheduling and allocation in distributed settings.

Acknowledgements: This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

1. K. Bhattacharya, R. Hull, and J. Su. A data-centric design methodology for business processes. In *Handbook of Research on Business Process Modeling, chapter 23*, pages 503–531, 2009.
2. M. Böhm. *Cost-based optimization of integration flows.* PhD thesis, 2011.
3. Conor Cunningham, Goetz Graefe, and César A. Galindo-Legaria. Pivot and unpivot: Optimization and execution strategies in an rdbms. In *VLDB*, pages 998–1009, 2004.

4. U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson. Data integration flows for business intelligence. In *Proc. of the 12th Int. Conf. on Extending Database Technology: Advances in Database Technology, EDBT*, pages 1–11. ACM, 2009.
5. D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. of the 1999 ACM SIGMOD Int. Conf. on Management of data*, pages 311–322, 1999.
6. F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.
7. Y.E. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. Zoo: A desktop experiment management environment. In *Proc. of 22th Int. Conf. on Very Large Data Bases VLDB*, pages 274–285, 1996.
8. G. Kougka and A. Gounaris. On optimizing work ows using query processing techniques. In *SSDBM*, pages 601–606, 2012.
9. D.T. Liu and M.J. Franklin. The design of GridDB: A data-centric overlay for the scientific grid. In *VLDB*, pages 600–611, 2004.
10. R. Minglun, Z. Weidong, and Y. Shanlin. Data oriented analysis of workflow optimization. In *Proc. of the 3rd World Congress on Intelligent Control and Automation, 2000 - Volume 4*, pages 2564 – 2566. IEEE Computer Society, 2000.
11. S. Narayanan, U. V. Catalyrek, T. M. Kurc, X. Zhang, and J.H. Saltz. Applying database support for large scale data driven science in distributed environments. In *Proc. of the 4th Workshop on Grid Computing*, 2003.
12. Maja Pesic, Dragan Bosnacki, and Wil van der Aalst. Enacting declarative languages using LTL: Avoiding errors and improving performance. In *Model Checking Software*, pages 146–161. 2010.
13. T. K. Sellis and A. Simitsis. Etl workflows: From formal specification to optimization. In *ADBIS*, pages 1–11, 2007.
14. S. Shankar, A. Kini, D.J. DeWitt, and J. Naughton. Integrating databases and workflow systems. *SIGMOD Rec.*, 34:5–11, September 2005.
15. A. Simitsis, P. Vassiliadis, and T. K. Sellis. State-space optimization of etl workflows. *IEEE Trans. Knowl. Data Eng.*, 17(10):1404–1419, 2005.
16. A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *Proc. of the 2012 ACM SIGMOD Int. Conf. on Management of Data*, pages 829–840, 2012.
17. U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proc. of the 32nd Int. Conference on Very large data bases VLDB*, pages 355–366, 2006.
18. V. Tziovara, P. Vassiliadis, and A. Simitsis. Deciding the physical implementation of ETL workflows. In *Proc. of the ACM 10th Int. Workshop on Data warehousing and OLAP DOLAP*, pages 49–56, 2007.
19. P. Vassiliadis and A. Simitsis. Near real time ETL. In *New Trends in Data Warehousing and Data Analysis*, pages 1–31. 2009.
20. Marko Vrhovnik, Holger Schwarz, Oliver Suhre, Bernhard Mitschang, Volker Markl, Albert Maier, and Tobias Kraft. An approach to optimize data processing in business processes. In *VLDB*, pages 615–626, 2007.
21. Ramana Yerneni, Chen Li, Jeffrey D. Ullman, and Hector Garcia-Molina. Optimizing large join queries in mediation systems. In *ICDT*, pages 348–364, 1999.
22. Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *SIGMOD Rec.*, 34:37–43, 2005.

Optimization of Data-intensive Flows: Is it Needed? Is it Solved?

Georgia Kougka
Aristotle University of Thessaloniki
georkoug@csd.auth.gr

Anastasios Gounaris
Aristotle University of Thessaloniki
gounaria@csd.auth.gr

ABSTRACT

Modern data analysis is increasingly employing data-intensive flows for processing very large volumes of data. As the data flows become more and more complex and operate in a highly dynamic environment, we argue that we need to resort to automated cost-based optimization solutions rather than relying on efficient designs by human experts. We further demonstrate that the current state-of-the-art in flow optimizations needs to be extended and we propose a promising direction for optimizing flows at the logical level, and more specifically, for deciding the sequence of flow tasks.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

data flow optimization; task reordering

1. INTRODUCTION

Nowadays, not only more and more data is produced, but there is also an increasing need for end-to-end processing of this data. End-to-end processing includes tasks, such as cleaning, extraction, integration and analytics, and as such, gives rise to data-intensive flows that go beyond traditional ETL (Extract-Transform-Loading) flows; the latter are restricted to simpler transformation task sequences and purpose, namely to populate a data warehouse. Data-intensive flows are encountered in both business intelligence [4] and scientific [13] settings.

Currently, data flows are typically designed manually, although commercial tools may provide some simple, static, cost-oblivious rule-based optimizations [6, 7]. Interestingly, there is an increasingly large portion of flow designers that are not IT experts [1], which raises doubts about the optimality of such manual designs. In addition, the optimality of a data flow depends on statistics, such as task costs and selectivities, which means that an optimal flow execution plan for a specific dataset may become sub-optimal when applied to another dataset with different statistics, or even for the same dataset if its characteristics evolve, as typically occurs in streaming real-time analytics. Because of all these factors, rather than relying on the skills of the designer, we need to resort to automated cost-based optimization solutions.

DOLAP'14, November 7, 2014, Shanghai, China.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-0999-8/14/11 ...$15.00.
http://dx.doi.org/10.1145/2666158.2666174 .

Flow execution can be defined both at *logical* and the *physical* level. At the logical level, the partial order of the tasks is typically represented as a directed acyclic graph (DAG), which describes the flow of the data from the source tasks to the sink ones and the exact sequence of tasks in between. Logical flow optimization bears similarities with database query optimization but the problem is actually more complex because (i) query optimizers cannot consider the dependency constraints between tasks that appear in data flows, (ii) the tasks in a flow execution graph do not necessarily belong to a specific set of operators with clear semantics, and (iii) the optimization objective is not limited to performance. Overall, data flow optimization can be inspired by query optimization techniques that perform structure reformations, such as reordering and introducing new tasks in an existing flow, but it cannot fully rely on them. For example, in [9] ad-hoc query optimization methodologies are employed for optimizing the flow execution plan by reordering and introducing filtering tasks. Other logical flow optimization proposals consider swapping re-orderable flow activities, merging tasks, splitting, and so on, in order to generate new flow execution plan alternatives for ETL flows [14]. Additional narrower proposals include task consolidation for reducing the overall execution time [16, 5].

At the physical level, a wide range of implementation aspects need to be specified so that the flow can be executed. The most significant of them include the choice of the exact implementation alternative for each task, the selection of the execution engine to run tasks, scheduling details, the manner in which data is transmitted between tasks, decisions as to whether the tasks are executed in a pipelined or step-wise fashion, and so on. For all those aspects, several techniques have been proposed, which assume that the flow has been already optimized at the logical level. For example, [17] proposes resource allocation algorithms and heuristic techniques taking into account constraints, such as cost optimization, user-specified deadline and workflow partitioning according to assigned deadlines, while a set of optimization algorithms for scheduling flows based on deadline and time constraints is analyzed in [2].

So far, the existing flow optimization methodologies tend to focus either on the physical or the logical level, without

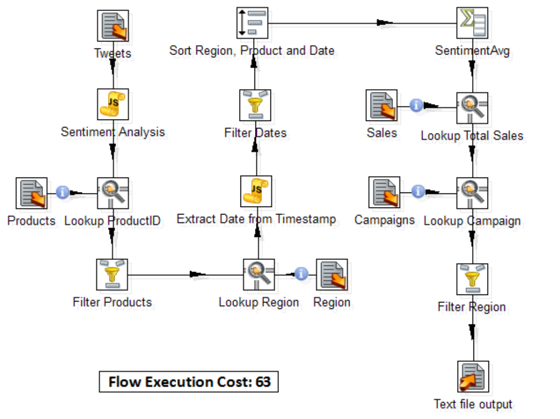


Figure 1: A real-world analytic data flow.

ID	*Flow Task*	*Cost(secs)*	*Selectivity*
1	Tweets (data source)	1.7	1
2	Sentiment Analysis	4.5	1
3	Lookup ProductID	5	1
4	Filter Products	1.9	0.9
5	Lookup Region	6.5	1
6	Extract Date from Timestamp	19.4	1
7	Filter Dates	2	0.2
8	Sort Region, Product and Date	173	1
9	SentimentAvg	10.3	0.1
10	Lookup Total Sales	10.8	1
11	Lookup Campaign	11.6	1
12	Filter Region	2	0.22
13	Report Output	1	1

Table 1: The cost and selectivities values.

providing a holistic optimization proposal for both levels. In this paper, we argue that the existing techniques regarding logical flow optimization, although they are interesting, they are inadequate. An optimal optimization solution referring at the physical level, which is based on a suboptimal logical plan, yields an overall suboptimal execution. We restrict ourselves to one of the simplest cases and we show that even for that case, the state-of-the-art can be significantly advanced. More specifically, we focus on *single-input single-output (SISO)* data flows, for which only the sequence of tasks needs to be specified so that all dependency constraints are respected and the single optimization criterion is the minimization of the sum of task execution times. We provide a real use case that demonstrates the need for more advanced optimization (Sec. 2), and we discuss the gap of existing optimization proposals and our proposal for near-optimal flow execution plans (Sec. 3).

2. MOTIVATIONAL EXAMPLE

A real data flow that processes free-form text data from Twitter commenting on products in order to compose a dynamic report that associates sales with marketing campaigns is shown in Figure 1. The flow is implemented with the help of the Pentaho Data Integration tool (`http://kettle.pentaho.com`). As shown in the figure, this data flow consists of 13 tasks (or nodes or activities) with a single streaming source that outputs tweets on products and a sink task where the resulting report is stored. During processing, it accesses four static sources (databases) through lookup operations. The remainder 7 tasks perform various operations, such as computing a single sentiment value for the products

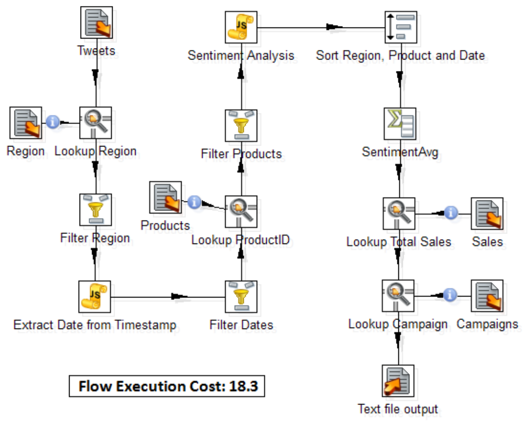


Figure 2: The optimized version of the data flow.

that are mentioned in the tweet (*Sentiment Analysis*), filtering according to several criteria including product type, and date, transformation of the timestamp text data to date and ad-hoc aggregation (*SentimentAvg* that averages sentiment values over each region, product, and date). At the final stage, the user has the option to narrow down the report in order to focus on a specific region. Table 1 shows the selectivity and cost values computed for a specific dataset of 1M records. We can observe that the most expensive tasks are the grouping and lookup tasks, the cost of which is up to two orders of magnitude compared to the less expensive ones. Also, there are 4 filtering tasks, while the rest do not modify the number of records (note that in general, selectivities may be higher than 1). For this flow, there are 38% precedence constraints (PCs, not presented in detail due to space constraints), where a fully constrained flow with n tasks and 100% PCs has $\frac{n(n-1)}{2}$ constraints and no equivalent ordering alternatives.

We run an exhaustive algorithm and we find the optimal flow for those statistics and precedence constraints, as shown in Figure 2. We also apply the best-performing approximate heuristic to date, which is proposed in [14]. Both exhaustive and approximate solutions are discussed in the next section. As we can see, the exhaustive optimization moves filtering according to region, which at the initial design has been placed at the end as a final optional step, at the very beginning for this specific flow due to the metadata in Table 1. A less obvious optimization is to move the pair of date extraction and filtering tasks upstream although the former is expensive and not filtering.

The total execution times of the initial, optimal and heuristic (i.e., approximately optimized) flow designs are 63, 18.3 and 36.5 seconds, respectively when run on a Intel Core i5 machine. This is a representative example of a manually designed data flow that exhibits significantly suboptimal behavior. In general, we can draw two observations. Firstly, optimal solutions may yield lower execution costs by several factors. A second equally important observation is that even in simple cases like the one examined here, existing heuristics may fail to closely approximate the optimal solution and generate the plan in Figure 2. The main reason in this example is that the approximate solution performs greedy swaps of adjacent activities; however the region filter cannot move earlier unless the campaign lookup task is moved earlier as well, an action that a greedy algorithm cannot cover.

3. OPTIMIZATION SOLUTIONS

Finding the optimal ordering of tasks is an NP-hard problem when (i) each flow task is characterized by its cost per input record and selectivity; (ii) the cost of each task is a linear function of the number of records processed and that number of records depends on the product of the selectivities of all preceding tasks (assuming independence of selectivities for simplicity); and (iii) the optimization criterion is the minimization of the sum of the costs of all tasks [3]. Here, we discuss the inherently non-scalable exhaustive solutions, the state-of-the-art heuristics and our proposal for flow optimization which improves on the latter heuristics.

3.1 Accurate (exhaustive) algorithms

Backtracking algorithm: The *Backtracking* algorithm, as presented in [8], finds all the possible execution plans generated after reordering the tasks of a given data flow preserving the precedence constraints (PCs). The algorithm enumerates all the valid sub-flow plans after applying a set of recursive calls on these sub-flows and runs in $O(n!)$ time.

Dynamic Programming (DP): The rationale of the *DP* algorithm extends its query optimization counterpart to calculate task subsets of size n based on subsets of size $n-1$. For each of these subsets, we keep only the optimal solutions, which are valid with regards to the PCs. The time complexity of *DP* is $O(n^2 2^n)$.

Topological Sorting (TS): Contrary to query optimization, we have found that enumerating all orderings that meet the dependency constraints is a viable option for flows with numerous PCs, although in the worst case the *TS* algorithm runs in $O(n!)$. Due to space limitations, we do not give full details, but as shown in the experiments below, a variant based on [15] can significantly outperform the two other exhaustive approaches mentioned above in terms of optimization time (overhead).

3.2 Approximate (heuristic) algorithms

The state-of-the-art heuristics for flow task ordering is summarized below. Preliminary results regarding their efficiency in optimizing flows is provided in [10].

Swap: The *Swap* algorithm, proposed in [14], compares the cost of the existing execution plan against the cost of the transformed plan, if we swap two adjacent tasks provided that the PCs are always satisfied. This check is performed for every pair of adjacent tasks. The complexity is $O(n^2)$. **GreedyI:** The *GreedyI* algorithm is based on a typical greedy approach, which builds the flow execution plan incrementally. In each step, it adds at the end of the partial plan the activity with the maximum rank $\frac{1-selectivity}{cost}$ among those for which all the prerequisite tasks have been already added. The time complexity is $O(n^2)$. **GreedyII:** The rationale of the *GreedyII* algorithm is similar to *GreedyI* apart from the fact that the construction of the optimized execution plan is right-to-left (i.e., from the sink to the source). The algorithm is presented in [12]. **Partition:** The *Partition* algorithm, in each step, detects the set of tasks that can be added based on the PCs. For that set, it exhaustively finds the optimal sub-solution, and then proceeds to the next set until all activities have been added. It runs in $O(n!)$ time in the worst case.

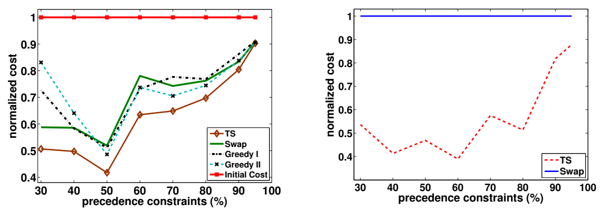


Figure 3: Average (left) and maximum (right) improvements of exhaustive solutions

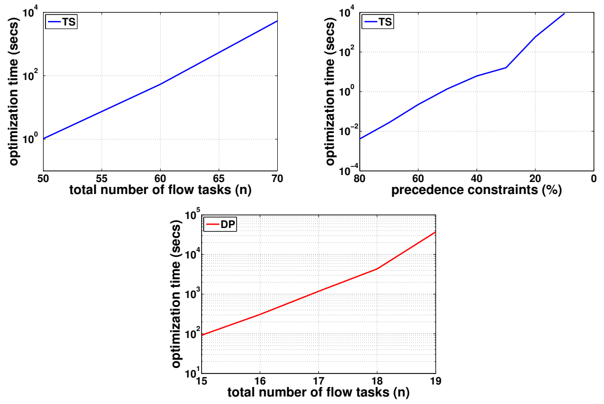


Figure 4: Optimization overhead for *TS* and *DP*

3.3 Pros and cons of exhaustive solutions

We conducted a set of experiments to further support the observation in Section 2 that the existing heuristics fail to approximate the optimal solution. We examined randomly generated data flows consisting of $n = 10, 15, \ldots, 25$ tasks, selectivity values $sel \in (0, 2]$ where selectivities higher than 1 denote increase in the task output, $cost \in [1, 10]$ and 30%-95% PCs. The flows were executed on an Intel Core i5 machine and all experiments were repeated 20 times. The results show that the performance improvement derived by the application of exhaustive algorithms is significantly high for small flows, as they provide the optimal solution. For example, Figure 3(left) presents the results for flows with 15 tasks; the accurate algorithms, such as *TS*, can have up to 59% better performance improvement compared to a random initial flow that just respects the PCs. On average, the best performing heuristic is *Swap*. Figure 3(right) shows the maximum normalized difference between *Swap* and *TS*, which can reach up to 61% in favor of the latter.

However, exhaustive solutions are inapplicable for medium and large flows, and/or few PCs. As shown in Figure 4(top), *TS* cannot scale in either the number of the flow tasks or the PCs. *DP*'s overhead does not depend on the PCs and, as shown in Figure 4(bottom), this algorithm is not practical for flows with more than 18 tasks. *Backtracking* is less scalable than *TS* as well, with higher overhead by a factor between 46 and 62 on average.

3.4 Our optimization proposal

In the quest of finding an efficient optimization solution for our problem, as our starting point, we chose the well-known KBZ query optimization algorithm for join ordering in [11]. That algorithm is of low complexity $O(n^2)$ and can consider PCs that can be represented as a rooted tree. The rationale is to consider the rank of each task and order tasks by their rank value; if this is not possible due to PCs, then tasks are merged and the rank values are updated accordingly. The evaluation results (not presented here in detail) show that, when applicable, this approach can be dozens of times less expensive than *Swap* and the other heuristics. However, allowing only tree-shaped PC graphs implies that there is no task with more than one independent prerequisite activity and the percentage of PCs is very low and decreases with the number of tasks (e.g., less than 10% for a 100-node flow); both cases do not occur frequently in practice.

So, to combine KBZ's efficiency and support for generic flows, we reduce our problem to that of transforming the DAG that typically represents PCs (an edge from one task to another denotes that the former must precede) into an acyclic one after removing edge directions. We initially propose a simple heuristic algorithm *RO-I* (for rank ordering) that, omitting implementation details, transforms the graph of PCs into an acyclic one by removing incoming edges with no maximum rank, if a task has more than one incoming edge. Then it applies the KBZ algorithm and is followed by a post-processing phase, where any resulting PC violations are resolved by moving tasks upstream if needed as prerequisites for other tasks placed earlier. That heuristic is simple but does not always behave well. We have investigated another approach, termed as *RO-II*, which detects paths in the PC graph that share an intermediate source and sink and merges them to a single path based on their rank values. In that way, both all PCs are preserved and the rationale of rank ordering is kept at the expense of implicitly examining fewer re-orderings. As such, these local optimizations do not guarantee a globally optimal solution.

Preliminary evaluation results are shown in Figures 5 and 6. We can see that the average improvements of *RO-II* over *Swap* can be significant, whereas *RO-I* in some cases outperforms *RO-II* but in others is significantly worse. For isolated runs, we have observed that *Swap* can be up to 7 times more expensive.

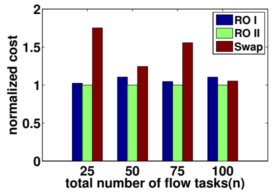


Figure 5: 20% PCs.

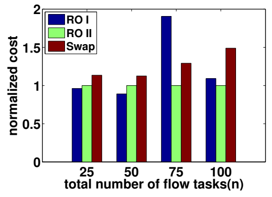


Figure 6: 50% PCs.

4. CONCLUDING REMARKS

The fact that existing logical optimization techniques are inadequate to provide a (near) optimal solution even for small flows implies that, even after applying the most advanced physical optimization techniques, the execution performance is suboptimal since the latter techniques depend on the former. More research is needed (i) for deciding the sequence of the flow tasks and (ii) for building more holistic approaches that consider additional factors, such as merging and splitting tasks and physical implementation details. For item (i), our proposal is promising and the results provided here provide strong insights in its ability to fill the gap between exhaustive non-scalable solutions and existing heuristics; however more research and thorough analysis and evaluation is needed for rank-ordering-based solutions.

Acknowledgments.
This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

5. REFERENCES

[1] D. Abadi *et al.* The beckman database research self-assessment meeting. Technical report, 2013.
[2] S. Abrishami, M. Naghibzadeh, and D. H. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, 29(1):158 – 169, 2013.
[3] J. Burge, K. Munagala, and U. Srivastava. Ordering pipelined query operators with precedence constraints. Technical Report 2005-40, Stanford InfoLab, 2005.
[4] S. Chaudhuri, U. Dayal, and V. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54:88–98, 2011.
[5] R. Dewan, A. Seidmann, and Z. Walter. Workflow optimization through task redesign in business information processes. In *HICSS*, pages 240–252. IEEE Computer Society, 1998.
[6] R. Halasipuram, P. M. Deshpande, and S. Padmanabhan. Determining essential statistics for cost based optimization of an etl workflow. In *EDBT*, pages 307–318, 2014.
[7] S. Holl, O. Zimmermann, M. Palmblad, Y. Mohammed, and M. Hofmann-Apitius. A new optimization phase for scientific workflow management systems. *Future Generation Comp. Syst.*, 36:352–362, 2014.
[8] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.
[9] G. Kougka and A. Gounaris. On optimizing work ows using query processing techniques. In *SSDBM*, pages 601–606, 2012.
[10] G. Kougka and A. Gounaris. Declarative expression and optimization of data-intensive flows. In *DaWaK*, pages 13–25, 2013.
[11] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, pages 128–137, 1986.
[12] N. Kumar and P. S. Kumar. An efficient heuristic for logical optimization of etl workflows. In *BIRTE*, volume 84 of *Lecture Notes in Business Information Processing*, pages 68–83. Springer, 2010.
[13] E. S. Ogasawara, D. de Oliveira, P. Valduriez, J. Dias, F. Porto, and M. Mattoso. An algebraic approach for data-centric scientific workflows. *PVLDB*, 4:1328–1339, 2011.
[14] A. Simitsis, P. Vassiliadis, and T. K. Sellis. State-space optimization of etl workflows. *IEEE Trans. Knowl. Data Eng.*, 17(10):1404–1419, 2005.
[15] Y. L. Varol and D. Rotem. An algorithm to generate all topological sorting arrangements. *The Computer Journal*, 24(1):83–84, 1981.
[16] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, and T. Kraft. An approach to optimize data processing in business processes. In *VLDB*, pages 615–626, 2007.
[17] Z. Xiao, H. Chang, and Y. Yi. Optimization of workflow resources allocation with cost constraint. In *Proc. of the 10th Int. Conf. on Computer supported cooperative work in design*, pages 647–656, 2007.

Future Generation Computer Systems 45 (2015) 133–148

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Practical algorithms for execution engine selection in data flows

Georgia Kougka *, Anastasios Gounaris, Kostas Tsichlas

Aristotle University of Thessaloniki, Greece

HIGHLIGHTS

- A set of anytime algorithms for yielding mappings of flow nodes to execution engines.
- An optimal solution with polynomial complexity for linear flows.
- Evaluation using both real and synthetic flows in a wide range of settings.
- Proof of the NP-hardness of the problem.

ARTICLE INFO

Article history:
Received 8 May 2014
Received in revised form
25 September 2014
Accepted 10 November 2014
Available online 26 November 2014

Keywords:
Data flows
Resource allocation
Heterogeneous machines
Anytime algorithms

ABSTRACT

Data-intensive flows are increasingly encountered in various settings, including business intelligence and scientific scenarios. At the same time, flow technology is evolving. Instead of resorting to monolithic solutions, current approaches tend to employ multiple execution engines, such as Hadoop clusters, traditional DBMSs, and stand-alone tools. We target the problem of allocating flow activities to specific heterogeneous and interdependent execution engines while minimizing the flow execution cost. To date, the state-of-the-art is limited to simple heuristics. Although the problem is intractable, we propose practical anytime solutions that are capable of outperforming those simple heuristics and yielding allocation plans in seconds even when optimizing large flows on ordinary machines. Moreover, we prove the NP-hardness of the problem in the generic case and we propose an exact polynomial solution for a specific form of flows, namely, linear flows. We thoroughly evaluate our solutions in both real-world and flows synthetic, and the results show the superiority of our solutions. Especially in real-world scenarios, we can decrease execution time up to more than 3 times.

1. Introduction

Our entry into the era of big data has signalled notable changes in the way scientific research is conducted and enterprises operate. More and more emphasis is put on processing large volumes of data in less, if not real, time in order to accomplish scientific or business intelligence tasks [1,2]. The most common approach to this end is to design and execute data flows, using workflow tools and platforms that take over the integration of multiple data sources, data manipulation and service orchestration.

Our work is largely motivated by the needs of modern business intelligence (BI) applications and data-intensive scientific workflows, e.g., in bio-informatics. Traditionally, BI builds on top of data-warehouses and data-marts, which are populated by periodic Extract–Transform–Load (ETL) flows. This setting has evolved in two ways. First, flows have become more complex encompassing text analytics and machine learning operations along with data transformation activities. In addition, they operate on both stored data and external, rapidly evolving runtime data, such as feeds and click streams. Second, flows are no longer executed on a single processing engine but their execution may span multiple engines; such flows are also referred to as hybrid flows [3]. Examples of execution engines include Hadoop clusters, traditional DBMS, R scripts and stand-alone tools, each of which may come in several different instances (e.g., both mysql and Oracle RDBMSs) or configurations (e.g., number of reducers in Hadoop) resulting in a big set of candidate execution platforms for executing a single flow (e.g., [4,3,5–8]).

We can follow two main approaches to executing data flows.[1] The first one involves the manual, low-level script-based design

* Corresponding author.
E-mail addresses: georkoug@csd.auth.gr (G. Kougka), gounaria@csd.auth.gr (A. Gounaris), tsichlas@csd.auth.gr (K. Tsichlas).

[1] Due to the increased impact of the volume of data in such flows, in the remaining part of the paper, we will use the terms workflows, data flows or simply flows interchangeably.

http://dx.doi.org/10.1016/j.future.2014.11.011
0167-739X/© 2014 Elsevier B.V. All rights reserved.

of flows, which are then executed in a step-wise fashion. Such an approach is prone to errors and sub-optimal execution, due to the complexity of the flows. The second approach views the workflows at a higher logical level and relies on flow optimizers to decide the technical execution details; this is akin to the role of optimizers in database systems. Optimizing data flows is a challenging multi-dimensional task; two of the most important dimensions include (i) the optimization of the structure of flows, which comes in a form of a directed acyclic graph, but its vertices do not necessarily have clear semantics, as is the case for relational operators; and (ii) the allocation of each of the flow vertices to a potentially different execution engine, choosing among multiple candidates.

Our work focuses on the second aspect mentioned above, and more specifically, aims to devise a mapping of flow nodes to execution engines so that the performance is maximized putting emphasis on keeping the optimization overhead low. The performance is measured in terms of the sum of the execution costs over all flow activities (or flow nodes). For this problem, only simple heuristic or non-scalable algorithms are known to date [4]; here we show how we can significantly improve upon the state-of-the-art. Moreover, we show how we can benefit from the existence of multiple execution engine options, rather than sticking to simple single-engine solutions. The main challenges of tackling this problem are posed by the following factors: the number of flow nodes and candidate engine or engine configurations may be large, the engines are heterogeneous in the sense that each engine is capable of executing a flow node in different time, and shipping data from one execution engine to another or switching between engines incurs cost, i.e., choosing the best execution engine for each flow node in isolation does not imply optimality [4].

Overall, we make the following contributions:

- We propose a set of anytime algorithms (Section 3) that, as shown in our experimental section, they are capable of yielding mappings of flow nodes to execution engines that are significantly better than naive approaches (Section 2), even when the flows are very large and our proposals are allowed to run only for a few seconds on an ordinary machine. These anytime algorithms fall into three main categories: branch and bound, random walk and set-cover ones.
- We propose an optimal solution with polynomial time complexity for the specific case, where the flow structure is linear, i.e., the flow is a chain of activities. Specifically, we present a polynomial dynamic programming algorithm that can yield exact solutions for linear flows and can act as an efficient approximate solver in more generic cases (Section 4).
- We evaluate our proposals using both real flows and synthetic in a wide range of settings. We declare winners among our proposals, depending on the type of the flow. In summary, the value of our solutions lies in that they are both effective in improving performance and easy to implement and light-weight. The dynamic programming approach performs remarkably well in many real-world settings and along with the anytime heuristics, we perform consistently better than current heuristics. The anytime proposals can run for any number of iterations tolerated by the users, e.g., to meet real-time constraints, and they are capable of yielding improved performance in short time. Especially when the flows are near-linear, as happens in many real-world cases, the execution cost can be decreased by more than 3 times. If the flows are completely linear, the improvements are even larger (Section 6).
- We prove the $\mathcal{NP}$-hardness of the problem at hand (which means that no solution with polynomial complexity can be found in the generic case) and at the same time it is impossible to approximate it within a small constant, unless $\mathcal{P} = \mathcal{NP}$ (Section 5).

2. Problem definition and background

In this paper, we will investigate resource allocation techniques, where each flow activity can run on multiple execution engines, of which, only one should be selected. At this point, we will not consider the optimization of the ordering of flow activities or the technical specifications of the available processors. To begin with, we represent the logical view of a flow as a directed acyclic graph (DAG), where each activity corresponds to a node in the graph and the edges between nodes represent intermediate data shipping among activities. Since we have different activity implementations for a specific engine or multiple engines, each flow activity has a processing cost in time units, which differs between engine instances or engine configurations. Additionally, data transfer from one engine to another and/or switching between engines has also a cost.

The main notation and assumptions of this Flow Activity Allocation problem (henceforth named FAA) are as follows:

- Let $G = (A, E)$ be a directed acyclic graph, where A denotes the nodes of the graph and E represents the data flow among the nodes, i.e., which activity feeds data to which activity.
- Let $A = \{a_1, \ldots, a_n\}$ be a set of (possibly streaming) activities of size n. Each flow activity is responsible for one or both of the following tasks: (i) reading or retrieving or storing data, and (ii) manipulating data. The definition of the activities and the complete flow G is left to the flow designer.
- Let $E = \{edge_1, \ldots, edge_{n'}\}$ be a set of edges of size n'. Each edge $edge_i$, $1 \leq i \leq n'$ equals to an ordered pair (a_j, a_k), so that $edge_i^{head} = a_j$ and $edge_i^{tail} = a_k$.
- Let $ENG = \{e_1, \ldots, e_m\}$ be a set of execution engines that activities can be allocated to; ENG's size is m. In general, the number of execution engines tends to be smaller than the number of flow activities. However, different engine instances and/or configurations (e.g., multiple Hadoop clusters, each with varying number of reducers) are essentially treated as different engines, so that the number of different engines at our disposal may well be larger than the number of the flow activities. Note that nowadays, it has become easier to support multiple execution engines for each activity; for example, in [5], it is discussed how a logical data flow activity definition can automatically be translated to several distinct physical implementations according to the underlying execution engine including SQL, pig-latin and PDI[2] scripts.
- Let $c_{i,j}$ be the execution time of an activity a_i when mapped to engine e_j. We assume that this information is available, through e.g., micro-benchmarking as in [6], and we do not deal with the engine configuration ourselves.
- Let $ce^{a_k}_{e_{i\to j}}$ be the cost associated with the graph edges. It consists of (a) the engine switching cost from engine e_i, which executes activity a_k, to engine e_j, which executes the subsequent activity; and (b) the data shipping from the output of activity a_k (executed on engine e_i) to the subsequent activity. The subsequent activity is the activity the edge points to. The first component depends on the two engines, while the second depends additionally on the data volume transferred across the edge; this volume depends on the sender activity. Overall, ce depends on the sender activity and the execution engines of the activities connected through the edge. As above, we assume that this metadata is available to our algorithms either through micro-benchmarking or through log files. We can support arbitrary settings of $ce^{a_k}_{i\to i}$ values denoting the edge cost for activities running on the same engine instance;

2 http://www.pentaho.com/product/data-integration.

however, in the remaining part, we assume that $ce^{a_k}_{i\to i}$ cost from engine e_i to engine e_i is 0, because there is no data transfer over the network and/or engine configuration changes involved and we will refer to the *ce* cost as *inter-engine* cost.

Our goal can be stated as the derivation of an allocation function $f : [1, n] \to [1, m]$, which expresses the mapping between activities and processor engines, so that (i) the *total execution time is minimized*; (ii) each activity is mapped to one and only one engine; and (iii) our allocation algorithms run in seconds at most. Generally, we denote the mapping between an activity a_i and a processor engine e_j as $f(i) = j$, where $1 \le i \le n$ and $1 \le j \le m$. The total execution time *TET* for a specific allocation is the sum of all the execution costs of each activity on their engines plus the cost of transferring data and switching between different engines. The latter occurs whenever two nodes of an edge belonging to *E* are allocated to distinct engines:

$$TET = \sum_{i=1}^{n} c_{i,f(i)} + \sum_{i=1}^{n'} ce^{edge_i^{head}}_{f(edge_i^{head})\to f(edge_i^{tail})}.$$

In a more generic scenario, there are constraints between allocations to denote the fact that not all activities can run on all engines. In the constrained case, the goal is to allocate all activities so that the total execution time cost is minimized subject to the allocation constraints.

As explained later, our solutions behave differently depending on whether the flows are linear or not. Linear flows are those that contain one and only one activity with no incoming edges and one and only one activity with no outgoing edges; all the other activities have exactly one incoming edge and one outgoing edge.

2.1. Motivational example

A real-world data flow, which has the role of analysing emerging temporal trends, is illustrated in Fig. 1. The data flow builds a taxonomy of current trends for a specific region, which are extracted from Twitter messages (tweets), and its purpose is to categorize the trends and derive key representative features.

This example flow comprises 14 activities for deriving the timestamp from tweets (*Extract timpestamp*), deriving the textual content (*Extract textual content*), performing look-up operations on auxiliary data sources (*LookupRegion*, *LookupTrends*), executing tasks that correspond to ordinary relational database operators (*Select, Join, Aggregation*), and performing analysis operators (*Qualitative Analysis*, *Label trends*, *Quantitative Analysis*).

In this example, we assume that 6 of the activities can execute on 2 candidate engines and 5 of the activities can execute on 3 candidate engines. The engines can be MapReduce engines, GPU accelerators and O-RDBMSs. The remaining 3 activities are performed using stand-alone scripts. In such a setting, the total number of different engine allocations in the figure is $2^6 3^5 = 15{,}552$. However, for each engine, there may be multiple engine instantiations (not shown in the figure). If, for example, there are 3 different instances per execution engine and stand-alone programs, there are more than $7.4 \cdot 10^{10} < 3^{14}(2^6 3^5)$ possible allocations. It is easy to see that this number increases exponentially in the number of available engines. Furthermore, moving data from one engine to another, e.g., from a Hadoop cluster to a database is associated with a time overhead. Also, not all activities can run on any engine; for example, the last activity can run only with the help of a GPU-based implementation or as a Map-Reduce program in a specific cluster. Our goal is to devise a concrete mapping of each flow activity to an execution engine in a small amount of time.

In the remaining part of this section, we present first, an exhaustive solution, upon which we later propose improvements, and second, heuristics that are fast albeit not very efficient in improving performance.

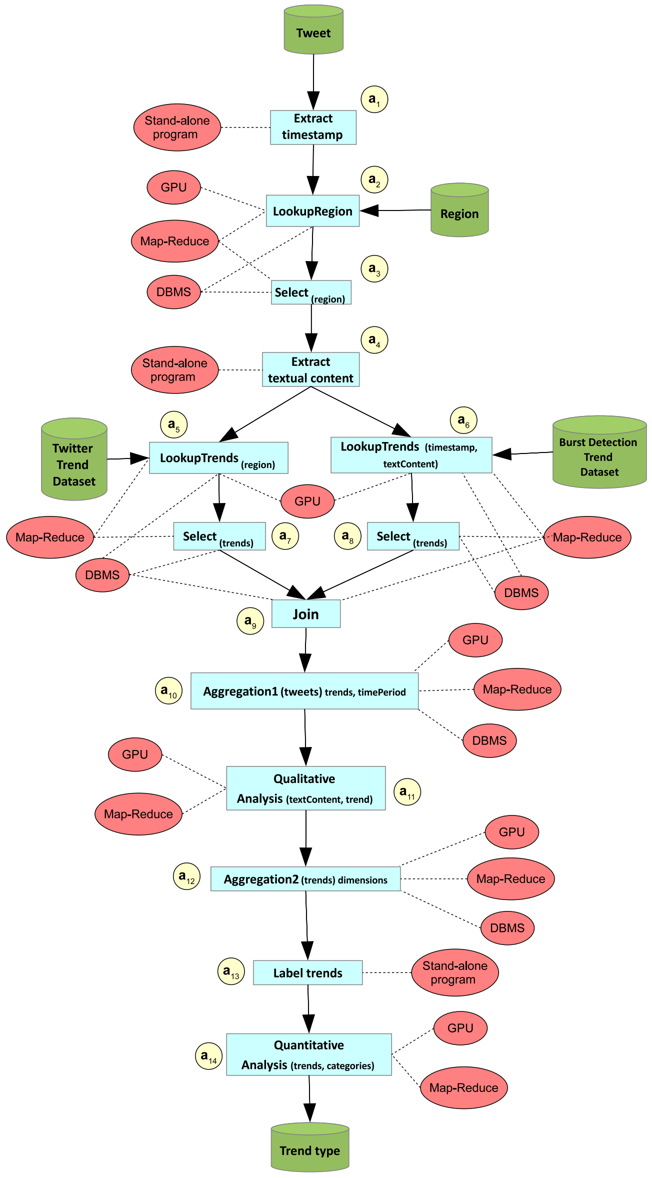


Fig. 1. A real-world data flow for interpreting emerging temporal trends.

2.2. An exhaustive solution

The rationale of an exhaustive methodology is to estimate all the possible combinations of engine allocations with regards to all flow activities. For a flow with n activities and m available execution engines, we have the following auxiliary matrices: (i) the $n \times m$ **C** matrix, where the element in the ith row and jth column is the $c_{i,j}$ execution time cost defined earlier; (ii) the $n \times m \times m$ **CE** matrix, where the element with the (k, i, j) coordinates is the $ce^{a_k}_{i\to j}$ inter-engine data shipping and engine switching cost; and (iii) the $n \times m$ **CONSTR** matrix, where the element in the ith row and jth column is set to 1 if the activity a_i can be mapped to the engine e_j.

The exhaustive algorithm iterates over all possible m^n allocations of execution engines to nodes. Due to this exponential complexity, it can be only applied to tiny flows, e.g., flows with very few nodes and candidate execution engines. In the exhaustive algorithm, each allocation is mapped to a distinct number in the range $[0, m^n - 1]$ with the help of the *mapNumberToAllocation* function. We can imagine each allocation plan as a number with n digits of base m. The value v of the ith digit from right to left denotes that the

Algorithm 1 Exhaustive Search

```
Require: $G(A, E)$, $ENG$, C, CE, CONSTR
 1: $f \leftarrow \emptyset$
 2: $mincost \leftarrow \infty$
 3: for all $i = 0 \ldots m^n - 1$ do
 4:   $f_{candidate} \leftarrow mapNumberToAllocation(i)$
 5:   if $f_{candidate}$ satisfies constraints in CONSTR then
 6:     $cost_{candidate} \leftarrow calculateCost(f_{candidate}, \mathbf{C}, \mathbf{CE})$
 7:     if $cost_{candidate} < mincost$ then
 8:       $mincost \leftarrow cost_{candidate}$
 9:       $f \leftarrow f_{candidate}$
10:     end if
11:   end if
12: end for
13: return an engine allocation plan $f$
```

ith activity is allocated to the engine $v + 1$. For example, the allocation number $(3521)_6$ denotes a mapping of a 4-node flow to engines, where $m = 6, f(1) = 2, f(2) = 3, f(3) = 6$ and $f(4) = 4$. Since the allocations with the smallest and the largest numbers are $(0000)_6$ and $(5555)_6$, respectively, all possible allocations are in the range of $[0, 6^4 - 1]$.

2.3. Heuristics

Another approach of allocation is to apply simple heuristics in order to avoid the complexity of estimating all the possible allocation combinations. For the purposes of this paper, we investigate two different heuristics, similar to those mentioned in [4]:

H1: this is a 2-step heuristic. First, we rank all engines based on their average execution cost for all flow activities in increasing order, i.e., the value for e_j is $\frac{1}{n}\sum_{i=1}^{n} c_{i,j}$. Then, we allocate each activity to the engine with the highest rank that is capable of executing that activity.

H2: this is also a 2-step heuristic. First, we rank all engines based on their execution cost for each flow activity separately. Then, we allocate each activity to the engine with the highest rank that is capable of executing that activity.

In Fig. 2, we show an example for a linear and a non-linear flow with $n = 5$ and $m = 3$. Because of the constraints between the engines, some allocations are not considered and the corresponding cells in C are shaded. Additionally, in this example we consider a CE matrix; for simplicity this matrix is $m \times m$ assuming that the values are the same for all n. For both flows, *H1* and *H2* provide a single allocation plan as shown in the figure; this is because the two flows differ only in their edges, which are not considered by naive heuristics.

In the remaining part of this work, we will refer to those solutions as simple or naive heuristics to distinguish them from our proposals in the next two sections.

3. Anytime algorithms

We now introduce anytime algorithms that can be stopped at any point and they are guaranteed to move closer to the optimal allocation the longer they are allowed to run. In a sense, the exhaustive algorithm can be classified as an anytime algorithm, too. But here, we present three types of solutions that are both efficient and effective, as verified by our experiments.

3.1. A branch and bound solution

A branch-and-bound (**BB**) approach can improve upon the naive exhaustive algorithm of the previous section. More specifically, we

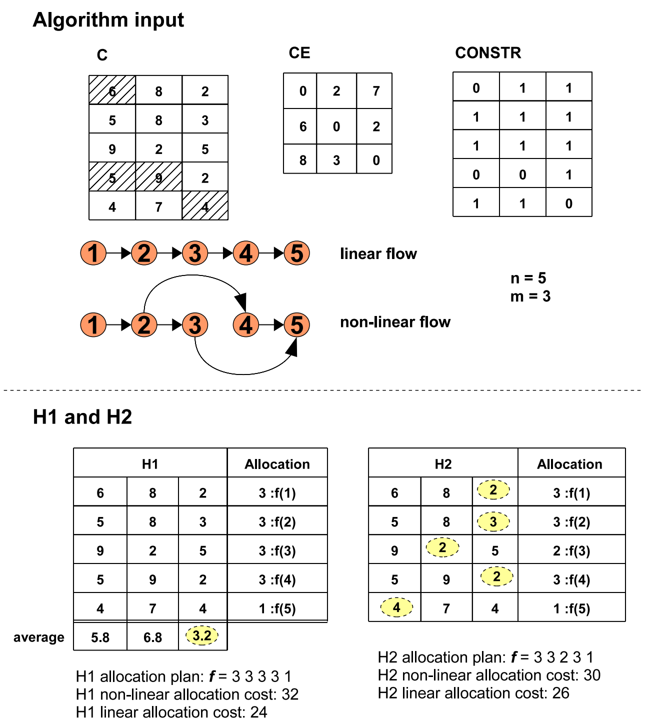


Fig. 2. An application of *H1* and *H2*.

can perform the following two main improvements. First, we can calculate the flow execution cost after each activity gets allocated and to abandon an intermediate allocation plan as soon as it exceeds the current minimum cost, which is the minimum of *H1* and *H2* algorithms for the first time. Second, when an allocation of an activity to a node is found not to satisfy the engine constraints, we move the allocation id counter as many steps as required in order not to examine a similarly invalid allocation of the same engine to the same node. For example, let us suppose that the $(3300)_6$ allocation is invalid because the 3rd node cannot run on engine 4. Instead of examining $(3301)_6$, $(3302)_6$, and so on (which are bound to be invalid as well), we move directly to $(3400)_6$.

Although, the two afore-mentioned improvements can yield speedups in the decision taking time, the computational complexity remains exponential, which renders the algorithm unsuitable for use in large flows. A remedy to this complexity problem is to cap the number of the allowed iterations. More specifically, we derive the **BB-IC** (Branch-n-Bound-Iteration Capping) algorithm, which, in addition to the two previous improvements allows only a pre-specified number of iterations (termed as *noi*). Given that threshold, the algorithm first estimates the maximum number $n_{\max}$ of nodes the allocation of which can be examined without exceeding the iteration threshold: $n_{\max} = \lfloor \log_m(noi) \rfloor$. Then, it runs the *H1* and *H2* and it keeps the best performing one. From the allocation plan of the best performing heuristic, it detects the $n_{\max}$ most expensive nodes, and investigates all their possible allocations using the branch-and-bound approach. The rationale behind this is to investigate other allocations for the parts of the flow that contribute the most to the total cost. The remaining nodes are allocated according to the allocation of the best performing heuristic.

Compared to Algorithm 1, the main changes are in three places: (i) the iteration of i is up to $m^{n_{\max}} - 1$ in line 3; (ii) $f_{candidate}$ corresponds to a plan with a subset of activities where the allocation of the remaining activities is defined by the best performing simple heuristic; as such, the cost estimation in line 6 needs to take this into account, and (iii) after line 11, we insert an else statement to increase the value of i, as explained above.

Algorithm 2 RWR-b

```
Require: $G(A, E)$, $ENG$, $\mathbf{C}$, $\mathbf{CE}$, $\mathbf{CONSTR}$, $length$, $r$
 1: $f \leftarrow best(H1, H2)$
 2: $mincost \leftarrow calculateCost(f)$
 3: for all $i = 1 \dots r$ do
 4:    $f_{candidate} \leftarrow f$
 5:    for all $j = 1 \dots length$ do
 6:       make a random change in $f_{candidate}$ that satisfies constraints in CONSTR
 7:       $cost_{candidate} \leftarrow calculateCost(f_{candidate}, \mathbf{C}, \mathbf{CE})$
 8:       if $cost_{candidate} < mincost$ then
 9:          $mincost \leftarrow cost_{candidate}, f \leftarrow f_{candidate}$
10:       end if
11:    end for
12: end for
13: return an engine allocation plan $f$
```

Algorithm 3 DP-cost

```
Require: $G(A, E)$, $ENG$, $\mathbf{C}$, $\mathbf{CE}$, $\mathbf{CONSTR}$
 1: for all $j = 1 \dots m$ do
 2:    if $f(1) = j$ satisfies constraints in CONSTR then
 3:       $DP_{cost}(1, j) \leftarrow C(1, j)$
 4:    end if
 5: end for
 6: for all $i = 2 \dots n$ do
 7:    for all $j = 1 \dots m$ do
 8:       if $f(i) = j$ satisfies constraints in CONSTR then
 9:          $k_{min} \leftarrow min_{1 \le k \le m}\{DP_{cost}(i-1, k) + CE(i, k, j)\}$
10:          $DP_{cost}(i, j) \leftarrow C(i, j) + DP_{cost}(i-1, k_{min})$
11:          $DP_{nodes}(i, j) \leftarrow k_{min}$
12:       end if
13:    end for
14: end for
```

3.2. Random-walk solutions

Our second approach to coping with the complexity of the problem is to explore the search space with random walks. We examine three main variants:

RW: Starting from the allocation derived from the best performing heuristic between *H1* and *H2*, we make random perturbations for a pre-specified number of times; this number is the *length* of the walk. In each iteration, we choose an activity in a round-robin fashion and we randomly alter its allocation.

RWR-r: This flavour extends the previous one by restarting the random walk *r* times. Each time, the starting point is a randomly selected allocation of all activities.

RWR-b: This flavour also employs restarts, but the starting point is the best performing allocation detected thus far (see Algorithm 2).

3.3. Dealing with large flows

For flows with large sets of activities and candidate engines, *BB-IC* and *RW* can explore only a very small part of the search space. E.g., in *BB-IC*, if $n = m = 100$ and $noi = 10,000$, then $n_{\max}$ is only 2. The two algorithms presented below, employ set-cover approaches in order to prune the search space; more specifically, they preprocess the candidate engines, and select a subset of *ENG* before applying the *BB-IC* and *RW* solutions. The intuition is that if, for large flows, we can derive a much smaller engine candidate set than the initial one, *BB-IC* and *RW* can improve their performance.

SC1: This set-cover based approach reduces the *ENG* set as follows. In each iteration, we count the number of activities each engine is allowed to execute, and we select the engine that is capable of processing the most activities. Conflicts are resolved arbitrarily. Then, we remove the activities supported by that engine and we proceed to the next iteration, unless there are no activities left. After we have selected the subset of engines, we apply both *BB-IC* and *RWR-b* (which run in very short time) and we choose the allocation with the lowest cost.

SC2: The *SC2* is another set-cover flavour, which takes into account the inter-engine cost. More specifically, it performs the first iteration exactly as *SC1* does. Then, in each subsequent iteration, it chooses the engine with the lowest average inter-engine cost with respect to the last added engine across all activities. Similarly to *SC1*, this procedure continues until all the activities can be executed on at least one engine, and the final allocation is found after applying both *BB-IC* and *RWR-b* to the reduced engine set.

The maximum number of iterations in the pre-processing engine selection phase for both set-cover approaches is *m*, but in practice, it is a small fraction of *m* and thus, the pre-processing step runs in a few milliseconds on a simple modern machine. Additionally, the number of iterations or restarts of BB-IC and RWR-b algorithms define the actual execution of the *SC* flavours that can be classified as anytime, too.

3.4. A hybrid solution

According to our experience, each of the previous anytime solutions may exhibit the best performance in different settings. Since all of them are lightweight and explore the search space in different ways, it is both possible and effective to run all of them and choose the best each time. Therefore, we introduce the **BEST** meta-heuristic that, after executing all *BB-IC*, *RWR-b*, *SC1* and *SC2*, chooses the one that yields the allocation plan with the lowest execution cost. As shown in Section 6, we can further increase the performance benefits by more than 10% because of that.

4. Dynamic programming

The previous proposals put emphasis on improving the naive heuristics without significantly raising the optimization overhead. Here, we propose a dynamic programming proposal that can find the optimal solution for linear flows and can act as an approximate solver for arbitrary flows.

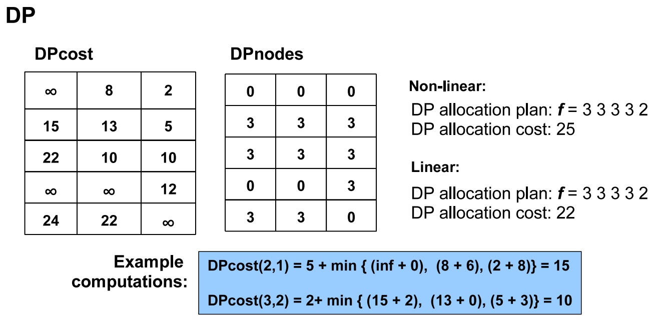
4.1. Detailed description

The rationale of the **DP** algorithm is to calculate the cost of increasingly larger portions of *A*, i.e., it starts of flows containing only a_1, then it examines flows containing (a_1, a_2), and so on, until it examines the complete flow. When examining flows with the first *i* activities, we consider the allocation costs of the flow consisting with the first $i-1$ activities. We employ a DP_{cost} matrix of size $n \times m$, where each cell (i, j) denotes the optimal cost of the plan with the first *i* activities when $f(i) = j$. The first row is initialized with the activity costs in $C[1, *]$. For the other rows, we have $DP_{cost}(i, j) = C(i, j) + \min_{k \in [1,m]}\{DP_{cost}(i-1, k) + CE(k, j)\}$. We also employ an auxiliary matrix, DP_{nodes}, which, in each cell (i, j), stores the engine for which the last part of the sum expression in the recursive formula is minimized. Overall, the last row of the DP_{cost} contains the costs when all activities are considered for all possible allocations of the last activity. In Algorithm 3, we show how the matrices are populated. The exact allocation is found by recursively examining the rows of the DP_{nodes} matrix from bottom to top (not included in the pseudocode).

When the flow is linear, the *DP* algorithm finds the optimal cost of an allocation; that cost is the minimum cost in the last row of DP_{cost}. To find the allocation function *f*, we start from the minimum value of the last row of DP_{cost}, the column of which denotes the allocation of the last activity $f(n)$; then, with the help of DP_{nodes}, we can recursively find the allocations $f(n-1), f(n-2), \ldots, f(1)$. Interestingly, the algorithm can be employed as an approximate solver for arbitrary flows. In that case, we can run the algorithm as previously and build the allocation plan, but such an allocation is not guaranteed to be optimal.

In Fig. 3, the allocation plan of *DP* for the metadata of example in Fig. 2 is the same for both flows as well, i.e., $f = (3, 3, 3, 3, 2)$. We can see that for both flows, *DP* yields a better solution than *H1* and *H2*. The allocation of *DP* is optimal for the linear case, but it is sub-optimal for the non-linear case.

Additionally, we should mention that for reasonable values of *noi*, *r* and *length*, *BB*, *BB-IC* and the random walk flavours find the optimal solution for both flows. The optimal solution of the non-linear case is $f = (3, 3, 2, 3, 2)$ with total cost 22 instead of 25.



DP

DPcost

∞	8	2
15	13	5
22	10	10
∞	∞	12
24	22	∞

DPnodes

0	0	0
3	3	3
3	3	3
0	0	3
3	3	0

Non-linear:
DP allocation plan: ***f*** = 3 3 3 3 2
DP allocation cost: 25

Linear:
DP allocation plan: ***f*** = 3 3 3 3 2
DP allocation cost: 22

Example computations:
DPcost(2,1) = 5 + min { (inf + 0), (8 + 6), (2 + 8)} = 15
DPcost(3,2) = 2+ min { (15 + 2), (13 + 0), (5 + 3)} = 10

Fig. 3. An application of *DP*.

4.2. Analysis

The time complexity of *DP* is $O(nm^2)$ because the size of the DP_{cost} matrix is $n \times m$ and in order to fill in each cell, the algorithm examines all m values of the cells in the previous row. As shown in the experiments, for a few hundreds of nodes and engines, the algorithm terminates in a few seconds. The space complexity is $O(nm)$, because of the $n \times m$ size of the DP_{nodes} matrix, which stores intermediate allocation. Note that, although we assume an $n \times m$ DP_{cost} matrix, we only need to keep two rows each time, thus the space complexity depends on DP_{nodes}. Below, we provide a sketch of the proof that *DP* is correct for linear plans; due to space limitations, we prove only that the cost found by *DP* is optimal.

Theorem 1. DP *finds the minimum cost of a linear flow.*

Proof. A sketch using induction on the size of the set A is as follows. If $n = 1$, the optimal solution is trivial and is found by the algorithm. Let the algorithm find the optimal solution $OPT(n, j)$ for $n = x$ and all engines $1 \le j \le m$. Assume now that $n = x + 1$. For the cost of the $(x + 1)$th running on e_j, the *DP* algorithm examines, for all possible allocations of the xth activity, the sum of the allocation cost of the first x activities and the inter-engine cost $ce^{x}_{f(x)\to j}$. The first part of that sum is optimal. The second part of the sum corresponds to the cost incurred by an edge $(x, x+1)$, which is the only real edge that exists between the first x activities and the $(x+1)$th one. Thus, for $n = x+1$, *DP* examines the whole set of valid combinations of optimal allocations of the first x activities plus the inter-engine cost between the first x activities and the $(x + 1)$th one, i.e., it does not miss any valid solution. □

5. Theoretical analysis

In the following we prove that the FAA (Flow Activity Allocation) problem is not only $\mathcal{NP}$-hard (the corresponding decision problem is $\mathcal{NP}$-complete) but at the same time it is impossible (unless $\mathcal{P} = \mathcal{NP}$) to approximate it within a small constant factor. The proof concerns a simplification of FAA, where $c_{i,j} = 1,\ 1 \le i \le n,\ 1 \le j \le m$ (uniform engines and activities with unit-processing times) while $ce^{a_k}_{i\to j} = 0$ when $i = j$ and $ce^{a_k}_{i\to j} = 1$, when $i \ne j$. We consider the case where the number of engines m can be arbitrary. In case where the number of engines is restricted we can get similar but slightly better results with respect to the approximation ratio bound. The proof is based on a transformation of the scheduling problem $P\infty|prec, c = 1, p_j = 1|C_{\max}$ (we use the notation introduced in [9] to denote scheduling problems). In this scheduling problem, the number of engines is arbitrary ($P\infty$), there are precedence constraints ($prec$) with unit-processing times for the activities ($p_j = 1$), there is a unit-time communication cost among engines ($c = 1$) and the goal is to minimize the makespan, that is the total length of the schedule. Note, that the simplified FAA problem could be represented as $P\infty|prec, c = 1, p_j = 1|\sum C_j$, since the goal is to minimize the total activity completion time. The following theorem is stated without a proof in [10], which we provide here for the sake of completeness.

Theorem 2. *The simplified FAA is $\mathcal{NP}$-hard and cannot be approximated by a polynomial-time algorithm with approximation error bound less than 8/7.*

Proof. [11] provides a polynomial-time transformation to prove that the decision scheduling problem $P\infty|prec, c = 1, p_j = 1|C_{\max}$ is $\mathcal{NP}$-complete. In particular, given an instance S of the 3-SAT problem, we construct an instance J for the scheduling problem. In a nutshell, for each variable in S, 6 activities are constructed and for each clause in S, 13 activities are constructed. Appropriate precedence constraints between these activities are enforced so that S has a truth assignment if and only if there is a schedule of J with makespan 6. This means, than in the case where S is a YES-instance of 3-SAT, then all activities in J are processed in the time interval [0, 6], while in the case where S is a NO-instance then in every possible feasible schedule of instance J there is at least one activity that completes at time 7 or later.

Let there be a polynomial-time approximation algorithm for the problem in the current paper with approximation $8/7 - \epsilon$, $\epsilon > \frac{19}{114k}$. We construct k copies of the instance $J, J_1, J_2, \ldots, J_k$, adding the precedence constraint that all activities in instance J_i are predecessors to all activities in $J_{i+1},\ 1 \le i \le k - 1$. Let the resulting schedule be denoted by J^*. If S is a YES-instance of 3-SAT, then instance J^* has a schedule with total activity completion time equal to $57mk^2$. This quantity is computed based on the precedence graph of the activities related to variables and clauses (see [11]). If S is a NO-instance of 3-SAT then the earliest possible time that J_i can start is $7i - 7$. Based on the precedence graph we get that at each integer time in the range $[7i-6, 7i-3]$ at most $4m$ activities can be completed. At time $7i-2$ at most m activities can be completed and finally at time $7i - 1$ at most $2m$ activities can be completed. As a result, the total completion time of J_i is at least $133mi-76m$, which when summed for all i, gives that the minimum total completion time for all activities in J^* is $66.5mk^2 - 9.5mk$.

From the above discussion we get that a polynomial-time approximation algorithm for our restricted problem with approximation ratio strictly less than $8/7 - \frac{19}{114k}$ is impossible unless $\mathcal{P} = \mathcal{NP}$, since this algorithm could be used to distinguish between the YES- and NO-instances of 3-SAT. This also proves the fact that the FAA problem is $\mathcal{NP}$-hard. □

6. Evaluation

In this section, we conduct a thorough evaluation of the solutions presented in the previous sections. We use both synthetic and

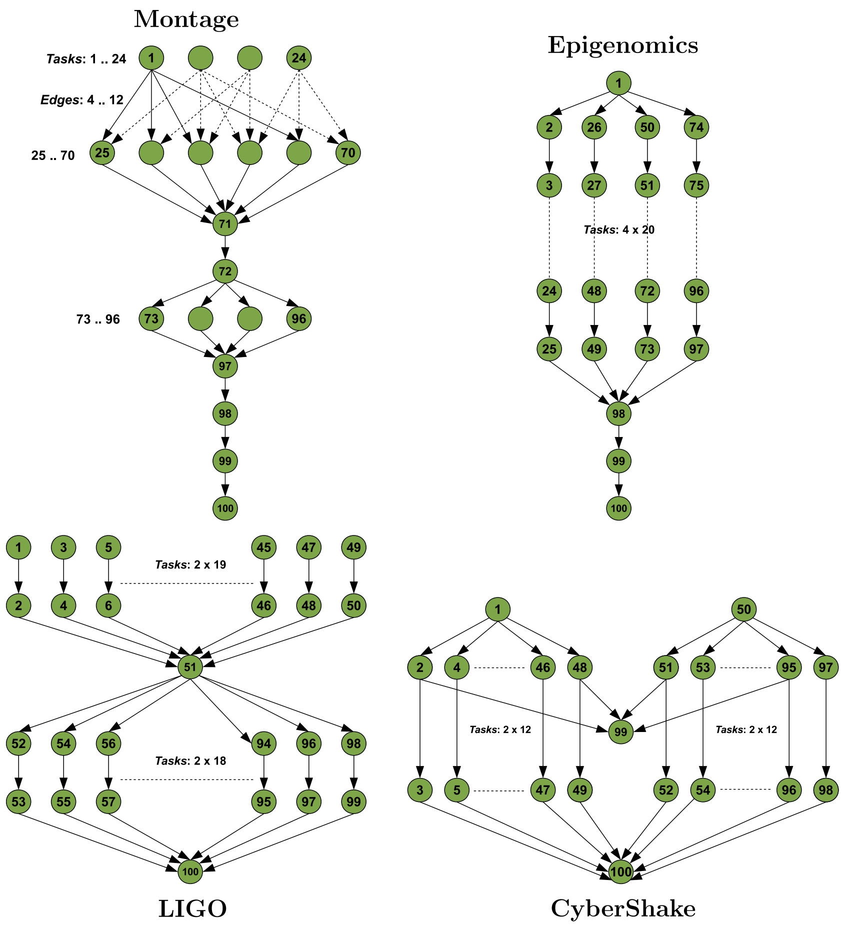


Fig. 4. The structure of the real workflows used in our experiments.

real-world flows. First, we examine real-world flows, where the focus is on the performance (Section 6.1.1) and testing under "real-world" conditions. For the latter, we examine two main aspects: the impact of inaccuracy in the statistical metadata (Section 6.1.2) and behaviour under settings where, intuitively, employing multiple engines does not seem promising; e.g., when all tasks can be executed on any engine, and the inter-engine costs are an order of magnitude higher than task processing costs (Section 6.1.3). The real-world flows are taken from [12]; they correspond to data-intensive scientific scenarios from several disciplines including astronomy, earthquake hazard characterization, biology and physics, and they are commonly used in the evaluation of techniques for data flows (e.g., [13]).

The purpose of the synthetic flows' experiments is to unveil the strengths and weaknesses of each allocation algorithm in random flow instances. In the synthetic flows, we focus on the following dimensions: (i) performance of the alternatives presented in terms of the estimated flow execution cost (Section 6.2.1); (ii) associated overhead of each solution in terms of real time spent in reaching allocation decisions (Section 6.2.2); (iii) accuracy, which refers to the deviation of the approximate flow execution costs compared to the optimal ones and is examined only when the optimal solution can be found in reasonable time (Section 6.2.3); and (iv) sensitivity analysis, which investigates the impact of different flavours and parameter values for the random walk proposals (Section 6.2.4).

In order to cover a wide range of settings, the flows vary in terms of the number of activities, engines, execution time of activities, transfer and switching costs between engines, and density of the DAG representing the flow. The probability of an engine to be capable of executing a specific activity is set to 50% unless otherwise stated. The activity and the inter-engine cost values are uniformly distributed in the range [1, 100]. To generate the inter-engine cost values we take into account the engine-independent amount of data outputted by each activity. The default settings of the random walk solutions are: number of restarts $r = 50$ and length of walk $l = 10^3$. The default setting of the number of iterations (*noi*) for the *BB-IC* solution is 10^4. These values are set in such a way that the anytime algorithms complete in a few seconds at most.

All the algorithms are implemented in MATLAB and the experiments were executed on a machine with an Intel Core i5 660 CPU and 6 GB of RAM. All experiments were repeated 50 times and we report the average values (except in Table 5).

6.1. Real-world flows

We experimented with 4 real-world flow structures, described in [12]. In particular, we created instances of the following flow types: Montage, Epigenomics, LIGO and CyberShake (see Fig. 4). For those flows, we experimented with $n = m = 100$ and the rest of the settings as in the introduction of this section. Initially, we assume that each activity processes the same amount of data and the inter-engine connection speed is the same for all pairs of engines; this implies that the inter-engine costs are activity-independent but we relax this assumption later.

6.1.1. Performance

In this experiment, we evaluate the relative performance in terms of execution time *TET* of the different policies (see Table 1). The numbers are normalized according to the execution cost yielded by *BEST*. For the montage flow, the *BEST* meta-heuristic is the best performing policy, while the naive heuristics *H1* and *H2*

Table 1
Normalized performance for real-world flows with 50% engine constraints.

Accurate statistics								
Flow	Algorithm							
	H1	*H2*	*DP*	*BB-IC*	*RWR-b*	*SC1*	*SC2*	*BEST*
Montage	1.3355	1.4083	1.4043	1.2555	1.2362	1.1578	1.0815	**1**
Epigenomics	1.5147	1.0282	**0.3208**	1.0057	1.0118	1.3652	1.1720	1
LIGO	1.3559	1.0512	**0.7601**	1.0245	1.0358	1.2604	1.0843	1
CyberShake	1.2858	1.1806	**0.9751**	1.1267	1.1304	1.1577	1.0489	1

Table 2
Normalized performance for real-world flows with 50% engine constraints and inter-engine cost activity-dependent.

Accurate statistics								
Flow	Algorithm							
	H1	*H2*	*DP*	*BB-IC*	*RWR-b*	*SC1*	*SC2*	*BEST*
Montage	1.3463	1.2507	1.2589	1.1427	1.0984	1.1443	1.0984	**1**
Epigenomics	1.9009	1.2167	**0.4664**	1.1390	1.0911	1.5464	1.3353	1
LIGO	2.0327	1.4644	**0.9433**	1.3526	1.2668	1.5341	1.3177	1
CyberShake	1.5200	1.3433	1.1649	1.1749	1.1308	1.2738	1.1899	**1**

Table 3
Normalized performance for real-world flows with 50% engine constraints.

Inaccurate statistics								
Flow	Algorithm							
	H1	*H2*	*DP*	*BB-IC*	*RWR-b*	*SC1*	*SC2*	*BEST*
10% inaccurate statistics								
Montage	1.0660	1.1054	1.1035	1.0373	1.4565	1.1331	1.2793	**1**
Epigenomics	1.4548	0.9874	**0.3107**	1.0000	1.8556	1.6620	1.7467	1
LIGO	1.3206	0.9844	**0.7026**	1.0000	1.7263	1.4803	1.5855	1
CyberShake	1.1728	1.0381	**0.8469**	1.0215	1.5245	1.2302	1.3635	1
30% inaccurate statistics								
Montage	1.1082	1.1542	1.1512	1.0649	1.5024	1.2354	1.3023	**1**
Epigenomics	1.4429	0.9789	**0.3034**	1.0000	1.8527	1.6091	1.7258	1
LIGO	1.2789	0.9890	**0.7188**	1.0000	1.7321	1.4334	1.5649	1
CyberShake	1.1614	1.0558	**0.8744**	1.0390	1.5639	1.2124	1.3491	1

yield 33% and 40% higher execution cost, respectively. However, the pattern changes for the rest of the real-world flow types. Those flows are not linear but they comprise linear subflows. So, *DP* outperforms the other policies. For Epigenomics, *DP*'s execution time is more than 3 times lower than those from the best performing heuristic, which is *H2*, and *BEST*. For LIGO and Cybershake, the DP's performance benefits are higher than 20% compared to the simple heuristics.

We now relax the assumption regarding the homogeneity of inter-engine network and the volume of data processed by each activity. More specifically, we experiment with scenarios where the inter-engine cost is a linear function of the data volume, and each activity may alter this volume by a factor uniformly drawn from 0.5 to 1.5 (denoting the pruning of half of the data and generating half as much additional data, respectively). The results are shown in Table 2. We observe that the heuristics yield relatively better performance for the montage flow, but the performance degradation with regards to *BEST* remains significant (25%). For the other three types of real-world flow structures, we observe that both *H2* and *DP* exhibit worse performance than the one reported in Table 1 and difference between our best performing proposal and the best performing heuristic widens. This is attributed to the fact that the more the heterogeneity in the cost associated with the graph edges, the more the need to consider these costs carefully, something that *H2* does not perform and *DP* performs only partially (since it considers only some of the existing edges). In the remaining part and in order to keep the evaluation concise, we will discuss mostly the case, where inter-engine costs are assumed to be activity-independent showing that even for that setting our solutions manage to yield improvements.

Regarding the time overhead, this is a couple of seconds even for the most time consuming techniques, such as the dynamic programming and random walk. Detailed experiments for the decision making overhead are presented later.

6.1.2. Imprecise statistical metadata

So far, we have assumed that the statistics in **C** and **CE** (the execution time and inter-engine shipping and switching costs) are accurately known. Here, we relax this assumption and we allow for imprecise statistics. We repeat the first part of the experiment in Section 6.1.1, but after we determine the allocation, we perturb the values in **C** and **CE** and we re-evaluate the total cost. In particular, we multiply each element in the two cost matrices with a scalar value $\alpha \in [0.9, 1.1]$ (denoting inaccuracies of $\pm 10\%$) or $\alpha \in [0.7, 1.3]$ (denoting inaccuracies of $\pm 30\%$). In this way, we emulate a situation, where the actual costs differ from those used during decision taking.

The results are shown in Table 3. For smaller inaccuracies of up to 10%, *DP* still outperforms the other policies for the last 3 flow types. The performance improvements vary from 15% up to more than 3 times. For Montage flows, *BEST* performs better, as in the case with no inaccuracies. However, the difference of *BEST* from the naive heuristics drops to 6.6%. When the inaccuracies grow larger, this difference is up to 10% for Montage flows. For such inaccuracies, *DP* clearly outperforms all the other policies for the other flow types.

Table 4
Normalized performance for real-world flows with no engine constraints.

Flow	Algorithm							
	H1	*H2*	*DP*	*BB-IC*	*RWR-b*	*SC1*	*SC2*	*BEST*
Inter-engine cost ∈ [1, 100] and no engine constraints								
Montage	1.0502	3.4899	3.42259	1.0307	1.0453	1.1453	1.0196	**1**
Epigenomics	1.0293	1.2458	**0.2980**	1.0018	1.0248	1.1788	1.0359	1
LIGO	1.0328	1.4764	**0.9932**	1.0023	1.0265	1.1653	1.0396	1
CyberShake	1.0645	2.2775	1.8019	1.0387	1.0615	1.1628	1.0288	**1**
Inter-engine cost ∈ [1, 1000] and no engine constraints								
Montage	1.4840	32.4686	30.3408	1.4739	1.4761	1.1429	1.0012	**1**
Epigenomics	7.0332	1.0136	**0.5915**	1.0012	1.0099	8.1907	7.1936	1
LIGO	1.2568	13.6752	7.7084	1.2489	1.2514	1.1293	1.0037	**1**
CyberShake	1.2459	21.2047	15.6062	1.2424	1.2429	1.1289	1.0002	**1**

6.1.3. *Settings discouraging multiple execution engines*

Intuitively, one might expect that when allowing any engine to run the complete flow (i.e., not having engine constraints), then not switching between engines, as in *H1*, yields the highest performance. However, as shown in the top part of Table 4, for the Epigenomics flow, *DP* still achieves more than 3 times lower execution cost. For the rest of the real-world flows, the improvements are significantly lower (between 3.2% and 6.4%). *H2* produces much worse results.

In addition, in real world, one might expect the inter-engine data transfer and switching costs to dominate. So, we perform another experiment, where the **CE** values are an order of magnitude higher than the values in **C**. The results are presented in the lower part of Table 4. Our solutions in that case improve the performance from 24.5% to 42%.

6.2. *Synthetic flows*

The results regarding the real-flows provide strong insights into the strengths of our solutions but they are tailored to the specific flows examined. To complement the evaluation, we randomly generate DAGs. The flows considered consist of $n = 10, 20, 50, 100, 200$ activities. We categorize the flows depending on their number of activities, as *small* (10 or 20 activities), *medium* (50 activities), *large* (100 activities) and *very large* (200 activities), based on the categorization in [14]. Regarding the exact shape of the flow graph *G*, we consider *dense* flows, where the probability of two activities to be connected with an edge is 50% (i.e., there exist $\frac{n(n-1)}{4}$ edges), *sparse* flows, where the edge probability is 20% (i.e., there exist $\frac{n(n-1)}{10}$ edges), and *linear* flows, where activity a_i is connected only with a_{i+1}.

The number of the available engines is $m = 10, 20, 50, 100, 200$. The Montage flow is the one closer to the random flow instances tested below. Also, the sparse flows are less sparse than the rest of the flows in Section 6.1.

As in the experimental setting of real-world flows, by default we assess the performance improvement where the inter-engine cost is activity-independent, but we later relax this to show that the our results hold for a wide range of inter-engine cost values.

6.2.1. *Performance improvement*

In the first set of experiments, we evaluate the flow performance in terms of flow execution time. We compare the two heuristics *H1* and *H2* against *DP*, *BB-IC*, *RWR-b*, *SC1*, *SC2* and *BEST*, which is the best among the last four. For the random walk flavours, we choose only the best performing one, and we leave their comparison for Section 6.2.4. The average results of these experiments are presented in Fig. 5. The numbers are normalized according to the execution cost yielded by *BEST* as previously.

Table 5
Maximum performance degradation of *H1* and *H2* in a single iteration compared to our solutions.

n	*m*				
	10	20	50	100	200
Dense flows					
10	2.92	2.78	3.16	3.77	4.37
20	3.03	2.96	3.46	4.38	3.65
50	3.23	2.88	3.01	3.09	4.68
100	2.41	2.32	2.58	2.80	3.09
200	1.89	2.39	3.58	3.83	2.56
Sparse flows					
10	2.10	2.67	2.24	2.31	2.80
20	2.70	2.22	2.67	3.41	2.90
50	3.15	3.01	2.67	2.62	3.10
100	2.60	3.50	2.75	2.42	2.68
200	3.27	2.68	2.42	2.25	2.24
Linear flows					
10	2.15	2.70	4.36	5.64	6.99
20	1.93	2.60	3.84	4.95	6.65
50	2.09	2.47	3.51	4.29	5.84
100	1.72	2.39	3.13	4.14	5.53
200	1.82	2.10	2.94	4.02	5.40

The main observation for dense and sparse flows is that the proposed anytime algorithms (i.e., *BB-IC*, *RWR-b*, *SC1*, *SC2* and *BEST*) consistently outperform the two simple heuristics; this is not the case for the *DP* proposal, which is proved to be optimal for linear flows and the more dense a flow is the higher the deviation of *DP*'s solution from the optimal one. Specifically, for dense flows (left column in Fig. 5), when the flow size is small, the best performing simple heuristic can run on average up to 100% longer than *BEST* (for $n = 10$ and $m = 100$). The best performing heuristic is *H1* because it implicitly tackles edge cost minimization contrary to solutions, such as *H2* and *DP*. The relative degradation decreases but remains significant as the flow size grows. For instance, the average degradation can be up to 70% for dense flows of medium size, 45% for large flows and 33% for very large flows. Note that the maximum performance degradation in a single iteration can be much higher, as shown in the upper part of Table 5. That table presents the highest number of times the best performing heuristic cost is higher than our best performing solution, which is always *BEST* for dense and sparse flows, and *DP* for linear flows. In a single iteration, the simple heuristics' execution costs observed are up to 368% larger for medium flows and up to 283% for very large flows.

In sparse data flows, the performance improvement is lower than in dense flows, but it is still considerable and up to 66%, 51%, 47% and 34% for small, medium, large and very large flows, respectively. We always compare our best performing solution against the best performing naive heuristic. Another observation is that both in dense and sparse data flows, *H1* outperforms the other heuristic *H2*, with some exceptions for small sparse flows.

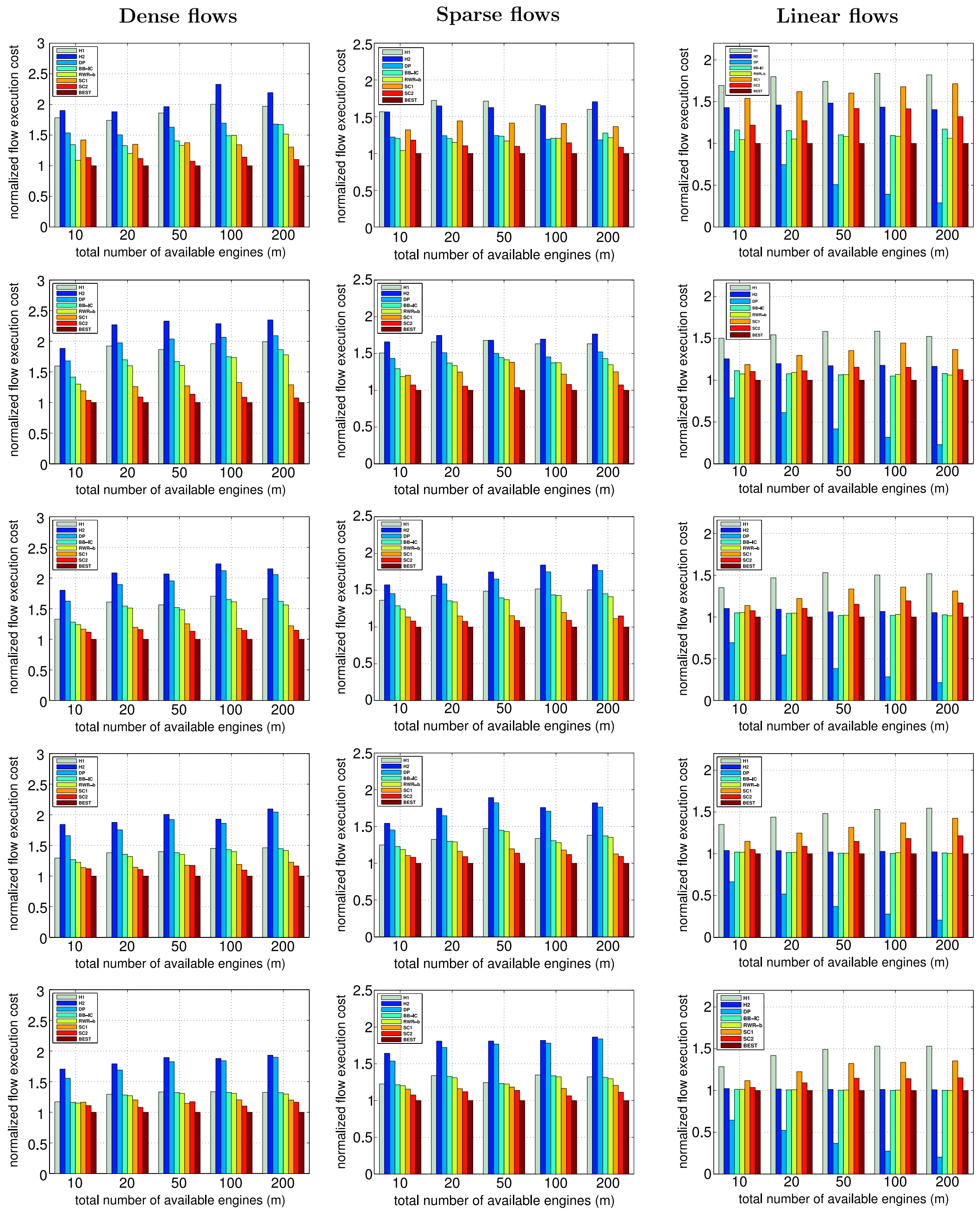


Fig. 5. Performance comparison when $n = 10$, 20, 50, 100 and 200 (from top to bottom).

As explained in Section 3, *BEST* is a meta-heuristic, which leverages *BB-IC*, *RWR-b* and set cover solutions. In general, the set cover solutions are the ones with the highest average performance between the approaches considered by *BEST* (apart from sparse flows when $n = m = 10$). Without *BEST*, our proposal with the highest performance would be at least 10% slower. Between *BB-IC* and *RWR-b*, which are used by all *SC1*, *SC2* and *BEST*, there is no clear winner, but in the majority of the settings, *RWR-b* is superior to *BB-IC*.

As far as the linear data flows are considered, the *DP* algorithm finds the optimal solution, and as such, achieves the lowest execution times. On average, *DP* can exhibit up to 7.5 times better performance than the naive heuristics. *BB-IC* and *RWR-b* attain similar performance improvements for large and very large flows. In all cases, both the *SC1* and *SC1* algorithms are outperformed by the *BB-IC* and *RWR-b* solutions. *H2*, which does not consider edge costs, performs better than *H1*, and in some cases better than some of our proposals, such as *SC1*.

In the next experiment, we show the performance of the algorithms when the average inter-engine cost becomes an order of magnitude lower than the average activity cost. More specifically, Fig. 6 depicts the execution cost of dense data flows when the inter-engine cost between engines $\in [1, 10]$. In this figure, we can see that, especially for non large flows, the naive heuristics perform very slightly worse than our solutions. This is expected since, when the inter-engine cost becomes zero, *H2* yields an optimal solution. However, even in this setting, the performance degradation for large and very large flows is significant and can reach 21%.

As for real flows, we relax the assumption and we consider activity-dependent inter-engine costs as in Section 6.1.1. The results are shown in Fig. 7 and confirm the conclusions that we discussed for real flows about the impact of execution engine homogeneity on the performance improvement of data flows.

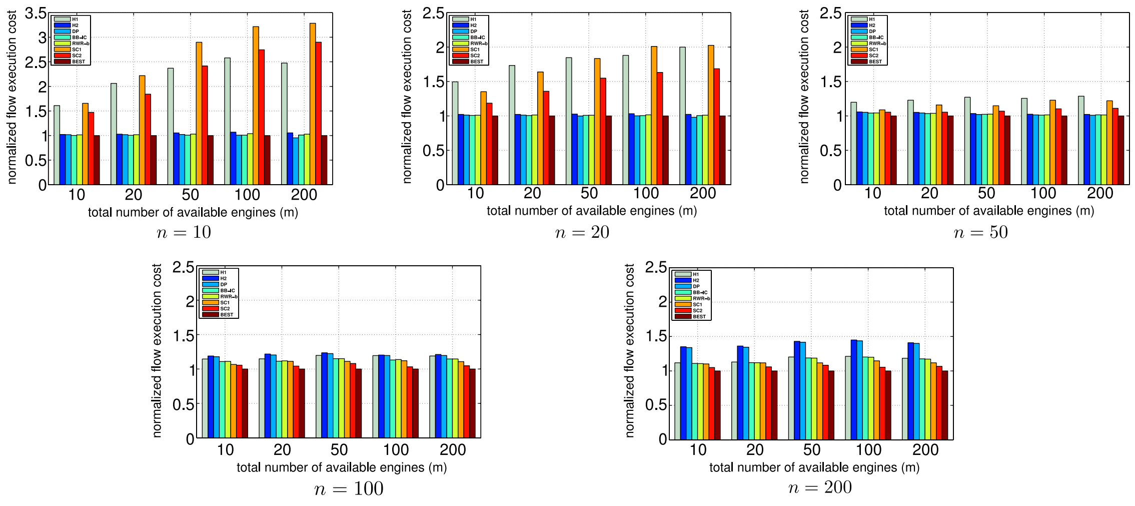


Fig. 6. Performance comparison when the inter-engine cost ∈ [1, 10] for dense flows.

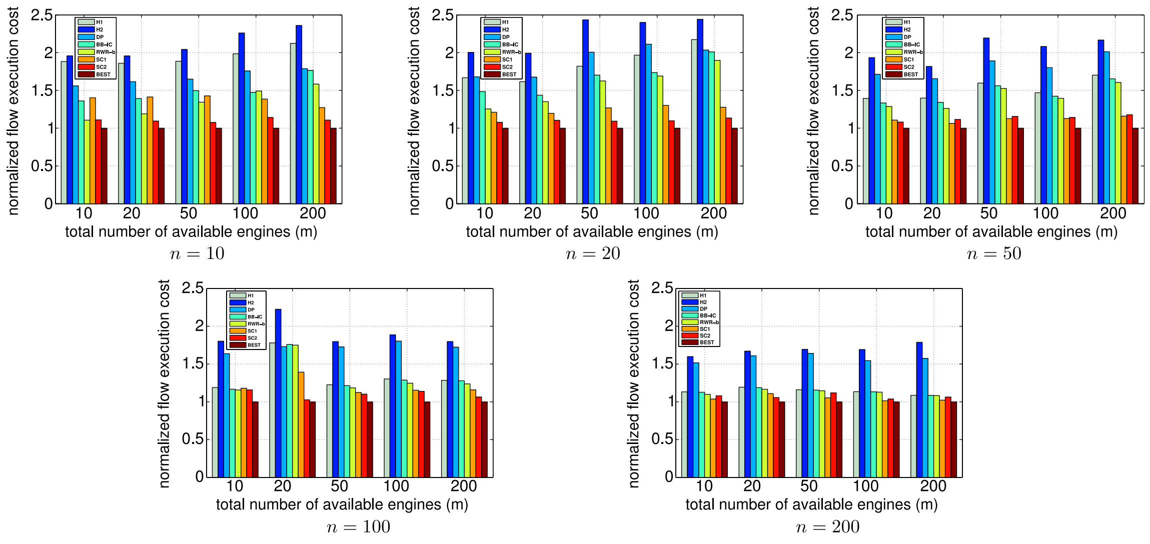


Fig. 7. Performance comparison when $n = 10, 20, 50, 100$ and 200 and the inter-engine cost is activity-dependent.

Specifically, we observe that the performance improvement of the data flows due to our proposals follows the same pattern for both inter-engine activity-dependent and activity-independent cases. We have observed that our solutions can be up to more than 5 times faster than the existing heuristics in isolated runs. Comparing Fig. 7 against the left column of Fig. 5, we see that *H1* is better than *H2* in Fig. 7, but its performance against our solutions is even worse for small and medium flows. For large and very large flows, the performance gap is slightly narrower, especially for a large number of candidate execution engines. In the remaining part, we will only discuss the case where the inter-engine costs are activity-independent for simplicity.

Figs. 8 and 9 refer to a more and a less constrained setting, where, on average, each flow activity can run on only 20% or 80% of the engines, respectively (instead of 50%). When having 20% engine probability, the performance degradation of the naive solutions is more evident for small flows (where it can be up to 51%), but becomes smaller for very large flows (where it can be up to 12%). In the case of 80% engine probability, the performance degradation increases compared to the results of 20% or 50% engine constraints. Specifically, for small flows, the simple heuristics in the best case are 63% worse than our proposals, while, in large flows, our average performance improvements are at least 60%.

6.2.2. *Decision making overhead*

We show the running time of the optimization process in Fig. 10. For simplicity, we discuss only dense flows, but the observations apply to all flow types. We can draw the following observations: the naive heuristics run in milliseconds for any size of flows and candidate engine sets. If the number of engines is up to 50, the *DP* and *BB-IC* algorithms run in hundreds of milliseconds. For $m = 100$, *DP* still runs in less than 1 s, except when $n = 200$. For $m = 200$, the average time overhead of *DP* is between 0.3 and 6.7 s.

RWR-b runs in 1 s for small flows, up to 1.8 s for medium flows; for large and very large flows, *RWR-b* does not exceed 4.2 and 13.7 s, respectively.[3] The overhead of the set cover solutions are largely determined by the overhead of *RWR-b*; it is slightly smaller than that of *RWR-b* since *SC1* and *SC2* examine a smaller set of engines. Overall, the running overhead is low, which supports our claim that our proposals are practical.

[3] The overhead of RWR-b is mostly due to the estimation of the cost of each allocation plan after each random change from scratch; for large flows, more efficient cost estimation approaches that reuse previous results can be devised.

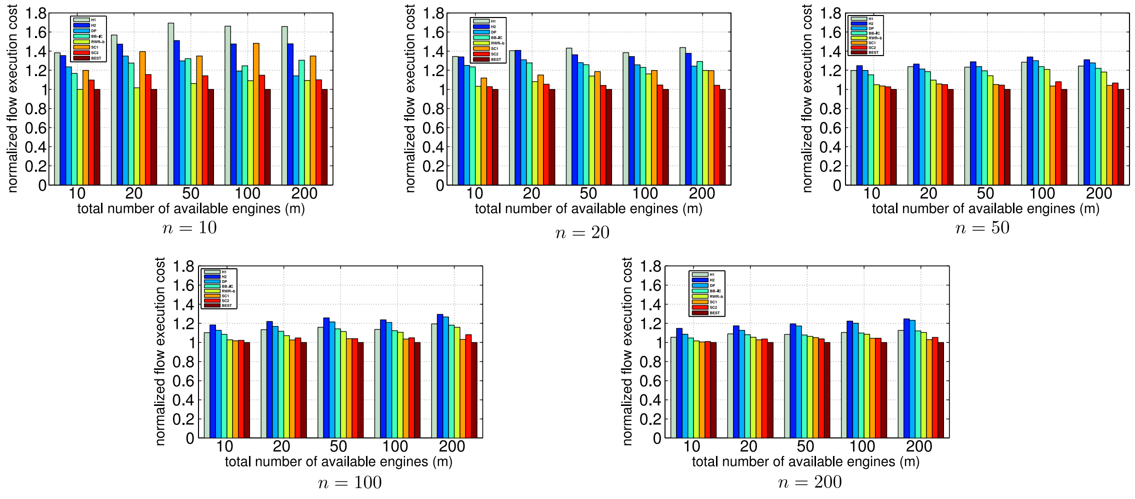


Fig. 8. Performance comparison when the probability of an engine to be capable of executing an activity is 20% for dense flows.

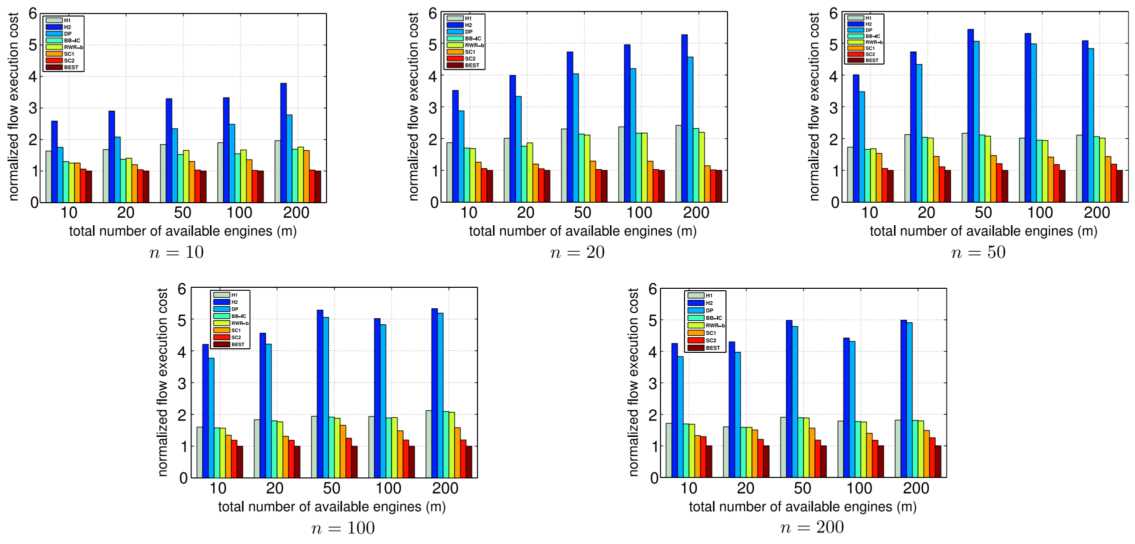


Fig. 9. Performance comparison when the probability of an engine to be capable of executing an activity is 80% for dense flows.

6.2.3. Accuracy

The accuracy of the algorithms can be accurately measured only if the optimal solution can be found. This can be done in reasonable time in two cases: (i) when the flows are linear; and (ii) when n and m are sufficiently small so that *BB* can be applied. For the first case, we can use the third column of Fig. 5, which shows the average performance of the algorithms. The accuracy of the anytime algorithms degrades as n and m increase. In addition, the bottom part of Table 5 shows the maximum performance degradation observed for the two simple heuristics.

We also check the accuracy of the algorithms for very small dense flows, where $n, m = 5, 6, 7, 8$. For such flows, *RWR-b* is remarkably accurate, and, on average, it is within 2% of the optimal solution provided by *BB*. The next more accurate algorithm is *BB-IC*, the average degradation of which is 15%. *DP* is 29% slower, whereas, the best performing naive heuristic, *H1* incurs 63% higher execution costs (see Fig. 11).

6.2.4. Random walk flavours

In Section 3, we discussed a set of random walk flavours and here we explain why we used only *RWR-b* in the previous experiments. We present the comparison of the flavours only for a representative setting: dense data flows with $n = 50$, $m = 50$. The results of this experiment are presented in Fig. 12. *RWR-b* algorithm has the best performance compared to *RW* and *RWR-r*, although the difference of performance and time overhead between *RWR-r* and *RWR-b* is negligible. Nevertheless, the optimization time of the simple *RW* is much lower than 1 s for activities at the expense of approximately 4.2% of performance degradation.

For the same experimental setting, we investigate the impact of the random walk length for *RWR-b*. We evaluate walk lengths of 10^3, 10^4 and 10^5, as shown in the middle row of Fig. 12. The main observation is that as we increase the length of the walks, the execution cost of the algorithm is slightly increased too, whereas the optimization time increases proportionally to the length of the walk. For large lengths the optimization overhead is on the orders of minutes without significant performance benefits. Finally, in the last set of experiment, we evaluate the impact of the number of restarts ($r = 10, 20, 30, 40, 50$), as shown in the bottom row of Fig. 12. According to the results, the impact of restarts does not significantly affect the performance: going from 10 restarts to 50 yields approximately 2% of improvement.

6.3. Summary of lessons learnt and discussion

The real-world flows in scientific scenarios tend to be sparse and either close to linear ones, or comprising many linear subflows.

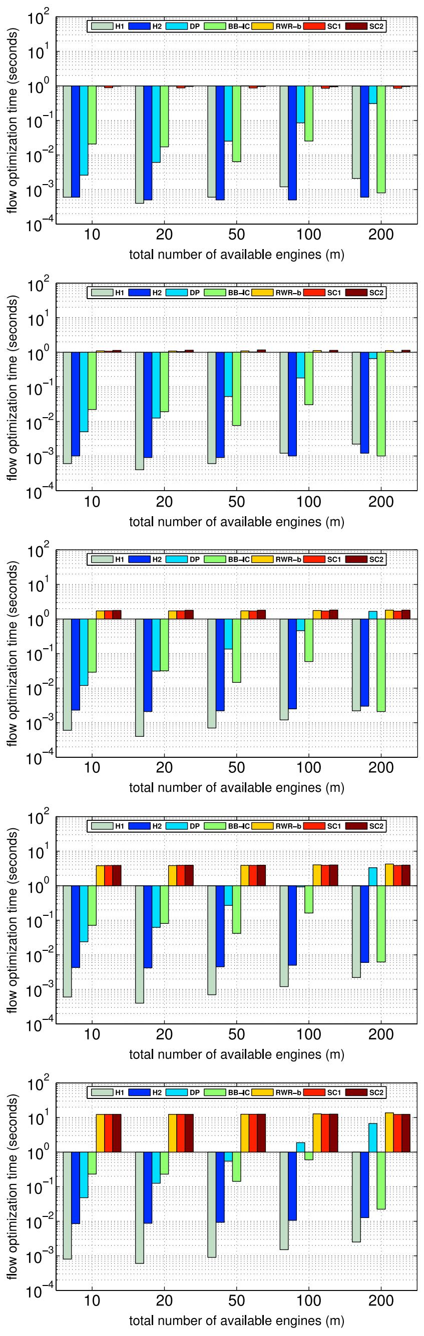


Fig. 10. Decision making time for $n = 10, 20, 50, 100$ and 200 (from top to bottom).

This implies that the contribution of the inter-engine cost to the flow execution cost is less significant compared to arbitrarily random flows. In scenarios, where there are many linear subflows, *DP* exhibits clearly better performance; otherwise *BEST* yields the lowest execution times. Our solutions can also yield significant benefits when there is no obligation to switch between engines (in the sense that an engine can run the complete flow) and the inter-engine costs are an order of magnitude higher than the processing costs.

From the experiments with the synthetic data, we can draw the following conclusions for random flows: (i) our solutions can efficiently handle even very large flows and outperform naive solutions; (ii) we can declare clear winners for different types of flows: for dense and sparse flows, *BEST* is the superior algorithm; for linear flows, *DP* is optimal; (iii) the performance improvements of our proposals compared to naive solutions are significant in every type of flows; and (iv) the running time of the decision making process completes in less than a second in most of the cases, which supports our claim that our proposals are practical.

Note that for random flows, *BEST* is a practical and efficient solution, and its efficiency is largely due to the set cover algorithms. For those algorithms, several additional flavours can be devised; for example to select engines according to their average inter-engine costs. Such flavours may support better specific scenarios, e.g., flow types where some tasks play the role of a hub with high degree of incoming and outgoing edges. In this work, we mostly focus on generic flows; the development of additional flavours tailored to specific flow structures is out of our scope.

7. Related work

The closest proposal to our work appears in [4,3], where the authors also deal with the complexity of flows in multi-engine environments and present a concrete workflow enactment system that supports hybrid flow execution. Apart from the system presentation, in [4], an exhaustive approach for allocating the activities of a flow to different execution engines is proposed in order to meet multiple-objectives, such as performance and fault-tolerance. In addition, heuristic techniques are presented for pruning the search space; those heuristics are equivalent to *H1* and *H2*. In our work, we improve upon such allocation schemes, and, through our evaluation, we show that our approaches are both scalable and significantly better than simple heuristics, when performance is the single optimization criterion.

[15] introduces an ant colony optimization algorithm that selects service instantiations between multiple candidates, in a setting where the flows mainly consist of a series of remote service invocations. In our work, we do not employ such type of algorithms, because their optimization overhead is at least two orders of magnitude higher (see indicative running times in [16]).

A state-of-the-art approach to flow scheduling is presented in [17,13]. Specifically, a set of optimization algorithms based on deadline and time constraints was analysed for scheduling flows. If we consider to adapt these methodologies in order to fit in our problem keeping only the allocation part regardless of deadlines, we will come to the conclusion that these methodologies are reduced to the simple heuristics presented in Section 2; more specifically the allocation part is reduced to *H2*, to which our proposals are shown to be superior. Another family of proposals aims at finding allocations of flow nodes to processors within a cluster, when the processors are homogeneous. Apart from that difference, which renders them inapplicable to our setting, typical assumptions are that there is no notion of inter-engine cost and there are no constraints with regards to the capabilities of an engine to execute a specific flow activity. Examples of such allocation approaches are described in [18–20].

For completeness, we briefly discuss additional aspects of flow optimization, which differ from our problem setting. [21] discusses optimal time schedules given a fixed allocation of activities to engines. Scheduling issues are also considered in works such as [22], which exploit existing systems, e.g. Pegasus, for task mapping procedure and [23], in which deadline constraints are taken into account. The proposals of [24–26] focus on methodologies of reordering and/or merging flow activities in order to yield improved performance, while keeping the flow semantics. In [27], flow activities are transformed in order to benefit from underlying data management infrastructures. [28,14] discuss optimization of data flows according to multiple objectives without considering engine allocation issues. In [29], a data oriented method for

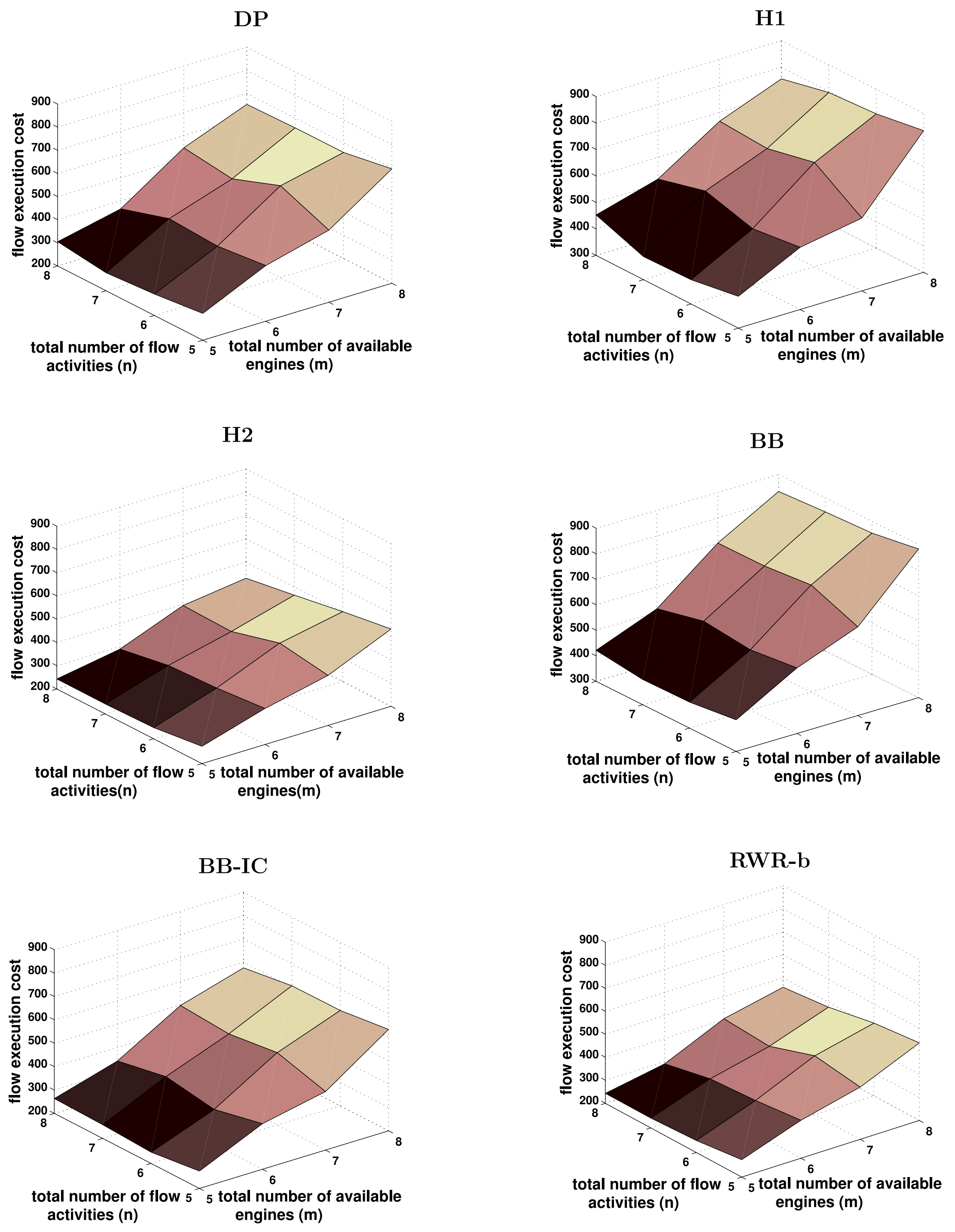


Fig. 11. Performance when $m = 5, 6, 7, 8$ and $n = 5, 6, 7, 8$.

workflow optimization is proposed in order to minimize execution cost. This method is based on the fact that data may be shared across several functions, and, as such, workflow performance stands to benefit from optimizations in the form of incorporating a shared database to handle common data-oriented tasks. Another proposal of flow optimization is presented in [30] based on soft deadline rescheduling in order to deal with the problem of fault tolerance in flow executions. In [31], an auction-based scheduling methodology for multi-objective flow optimization is presented; in our setting, choosing the most inexpensive engine is similar to the policy of the naive heuristic *H2*. Also, a methodology for minimizing the performance fluctuations that might occur by the resource diversity is proposed in [32]. Their proposal focuses on the delay correction during task execution. All these optimization aspects are orthogonal to our research.

The optimization of flows bears also similarities to distributed query optimization [33] and optimization of queries with user-defined functions [34]; however, in those problems, the focus is on the shape of the query plan and the ordering of the distributed operators (e.g., [34,35]) instead of deciding the mapping of a plan node to a specific engine. Other issues that differentiate query and flow optimization include the definition of the semantics of flow nodes, algebraic re-writing of flow plans and respecting inter-task dependencies.

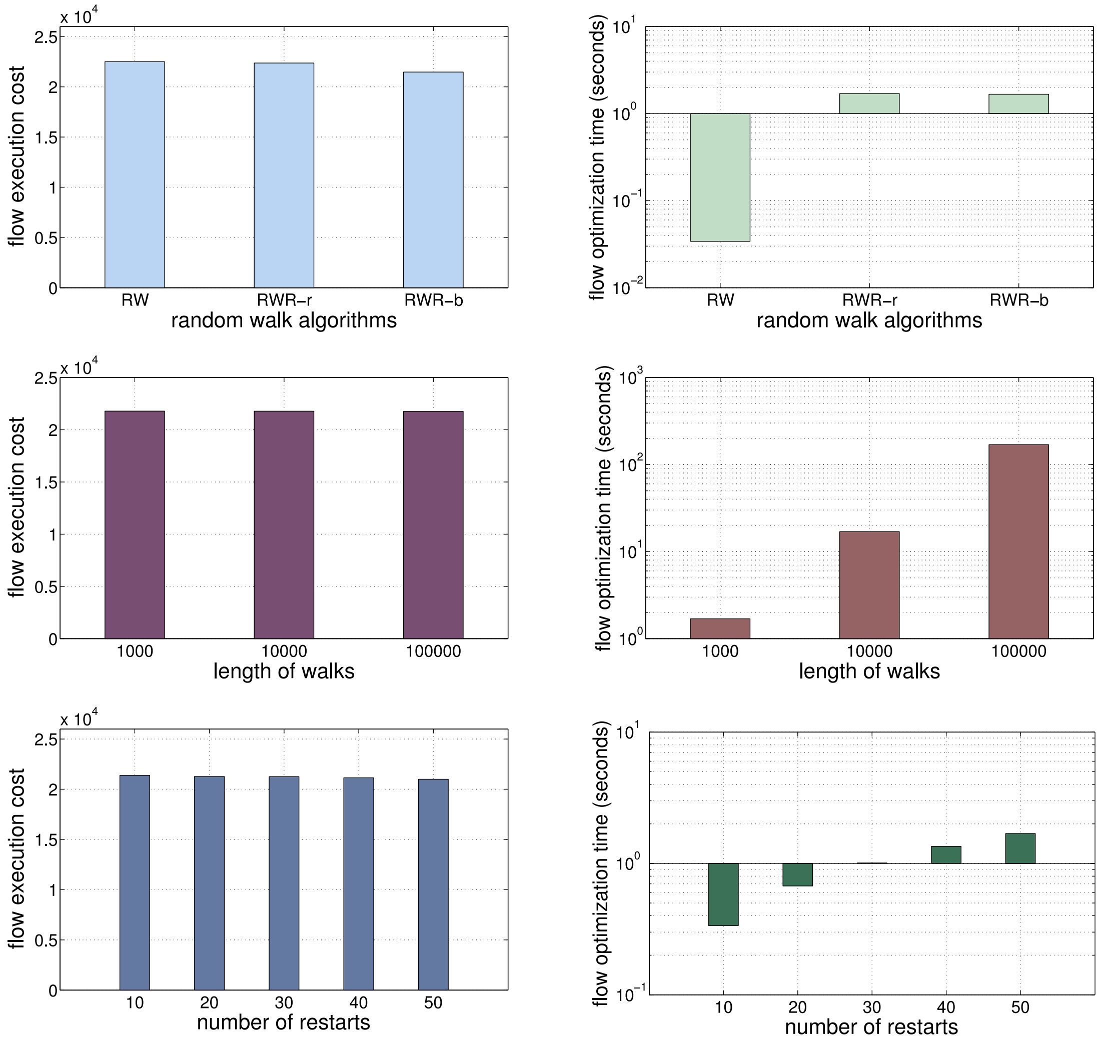


Fig. 12. 1st row: performance and optimization time of random walk flavours. 2nd and 3rd rows: impact of random walk length and restarts on *RWR-b*, respectively.

8. Conclusions and future work

In this work, we investigate the problem of allocating nodes of data-intensive flows to concrete executing engines. We prove that this problem is $\mathcal{NP}$-hard and cannot be approximated within a small constant. Due to the problem complexity, to date this problem is addressed using naive heuristics. In this work, we show that we can do significantly better without much overhead. We propose an optimal polynomial time dynamic programming solution for the specific case of flows that are linear, i.e., a chain of activities. Furthermore, we propose anytime algorithms that can handle any type and size of flows. With the help of our thorough experimentation, we declare clear winners depending on the type of the flow. Our proposals are capable of yielding solutions that are significantly better than naive approaches; actually, in real-world flows and conditions, they outperform those naive approaches by a factor of up to three. Our proposals are also easy to implement and are light-weight.

This work aims to propose fast algorithms for engine selection and focuses on the generic properties of the solutions. Apart from devising tailored solutions for each type of real-world flow, in the future, it is interesting to investigate solutions without the constraint of finding an allocation in a limited time period. Two further avenues for extending this work are to consider multiple objectives (e.g., both total time and makespan) and consider the impact of co-allocating activities to the same engine on the costs of those activities.

Acknowledgements

This research has been co-financed by the European Union (European Social Fund—ESF) and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF)—Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

We are also grateful to Alkis Simitsis for his insightful comments in early drafts of this work.

References

[1] T. Hey, S. Tansley, K. Tolle, The Fourth Paradigm: Data-Intensive Scientific Discovery, 2009.
[2] S. Chaudhuri, U. Dayal, V. Narasayya, An overview of business intelligence technology, Commun. ACM 54 (2011) 88–98.
[3] A. Simitsis, K. Wilkinson, U. Dayal, M. Hsu, HFMS: managing the lifecycle and complexity of hybrid analytic data flows, in: ICDE, 2013.
[4] A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal, Optimizing analytic data flows for multiple execution engines, in: SIGMOD Conference, 2012, pp. 829–840.
[5] P. Jovanovic, A. Simitsis, K. Wilkinson, Engine independence for logical analytic flows, in: Proc. ICDE, 2014.
[6] B. Huang, S. Babu, J. Yang, Cumulon: optimizing statistical data analysis in the cloud, in: SIGMOD Conference, 2013, pp. 1–12.
[7] G. Teodoro, T.D.R. Hartley, Ü.V. Çatalyürek, Renato Ferreira, Optimizing dataflow applications on heterogeneous environments, Cluster Comput. 15 (2) (2012) 125–144.

[8] A.D. Popescu, D. Dash, V. Kantere, A. Ailamaki, Adaptive query execution for data management in the cloud, in: CloudDB, 2010, pp. 17–24.
[9] R. Graham, E. Lawler, J. Lenstra, A. Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, in: Discrete Optimization II Proc. of the Advanced Research Inst. on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium, 1979, pp. 287–326.
[10] H. Hoogeveen, P. Schuurman, G. Woeginger, Non-approximability results for scheduling problems with minsum criteria, in: Integer Programming and Combinatorial Optimization, 1998, pp. 353–366.
[11] J.A. Hoogeveen, J.K. Lenstra, B. Veltman, Three, four, five, six, or the complexity of scheduling with communication delays, Oper. Res. Lett. 16 (3) (1994) 129–137.
[12] G. Juve, A.L. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Future Gener. Comput. Syst. 29 (3) (2013) 682–692.
[13] S. Abrishami, M. Naghibzadeh, D.H. Epema, Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds, Future Gener. Comput. Syst. 29 (1) (2013) 158–169.
[14] A. Simitsis, K. Wilkinson, U. Dayal, M. Castellanos, Optimizing ETL workflows for fault-tolerance, in: ICDE, 2010, pp. 385–396.
[15] W.N. Chen, J. Zhang, An ant colony optimization approach to a grid workflow scheduling problem with various QoS requirements, IEEE Trans. Syst. Man Cybern. Part C Appl. Rev. 39 (1) (2009) 29–43.
[16] Z. Zhang, X. Li, Based on TSP problem the research of improved ant colony algorithms, in: Electrical Engineering and Control, in: Lecture Notes in Electrical Engineering, vol. 98, 2011, pp. 827–833.
[17] S. Abrishami, M. Naghibzadeh, D.H.J. Epema, Cost-driven scheduling of grid workflows using partial critical paths, IEEE Trans. Parallel Distrib. Syst. 23 (8) (2012) 1400–1414.
[18] E. Schikuta, H. Wanek, I. Ul Haq, Grid workflow optimization regarding dynamically changing resources and conditions, Concurr. Comput.: Pract. Exp. 20 (2008) 1837–1849.
[19] M. Rahman, M.R. Hassan, R. Ranjan, R. Buyya, Adaptive workflow scheduling for dynamic grid and cloud computing environment, Concurr. Comput.: Pract. Exp. 25 (13) (2013) 1816–1842.
[20] R. Duan, R. Prodan, T. Fahringer, Performance and cost optimization for multiple large-scale grid workflow applications, in: Proc. of the ACM/IEEE Conf. on Supercomputing, 2007, pp. 1–12.
[21] K. Agrawal, A. Benoit, L. Magnan, Y. Robert, Scheduling algorithms for linear workflow optimization, in: IEEE IPDPS'2010, pp. 1–12.
[22] V.S. Kumar, P. Sadayappan, G. Mehta, K. Vahi, E. Deelman, V. Ratnakar, J. Kim, Y. Gil, M. Hall, T. Kurc, J. Saltz, An integrated framework for parameter-based optimization of scientific workflows, in: HPDC, ACM, 2009, pp. 177–186.
[23] Y. Yuan, X. Li, Q. Wang, X. Zhu, Deadline division-based heuristic for cost optimization in workflow scheduling, Inform. Sci. 179 (2009) 2562–2575.
[24] A. Simitsis, P. Vassiliadis, T.K. Sellis, State-space optimization of ETL workflows, IEEE Trans. Knowl. Data Eng. 17 (10) (2005) 1404–1419.
[25] G. Kougka, A. Gounaris, Declarative expression and optimization of data-intensive flows, in: DaWaK, 2013, pp. 13–25.
[26] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, K. Tzoumas, Opening the black boxes in data flow optimization, Proc. VLDB 5 (11) (2012) 1256–1267.
[27] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, T. Kraft, An approach to optimize data processing in business processes, in: VLDB, 2007, pp. 615–626.
[28] U. Dayal, M. Castellanos, A. Simitsis, K. Wilkinson, Data integration flows for business intelligence, in: Proc. of EDBT, 2009, pp. 1–11.
[29] R. Minglun, Z. Weidong, Y. Shanlin, Data oriented analysis of workflow optimization, in: Proc. of the 3rd World Congress on Intelligent Control and Automation, 2000—Vol. 4, IEEE Computer Society, 2000, pp. 2564–2566.
[30] K. Plankensteiner, R. Prodan, Meeting soft deadlines in scientific workflows using resubmission impact, IEEE Trans. Parallel Distrib. Syst. 23 (5) (2012) 890–901.
[31] H. Fard, R. Prodan, T. Fahringer, A truthful dynamic workflow scheduling mechanism for commercial multicloud environments, IEEE Trans. Parallel Distrib. Syst. 24 (6) (2013) 1203–1212.
[32] R.N. Calheiros, R. Buyya, Meeting deadlines of scientific workflows in public clouds with tasks replication, IEEE Trans. Parallel Distrib. Syst. 99 (PrePrints) (2013) 1.
[33] D. Kossmann, The state of the art in distributed query processing, ACM Comput. Surv. 32 (4) (2000) 422–469.
[34] S. Chaudhuri, K. Shim, Optimization of queries with user-defined predicates, ACM Trans. Database Syst. 24 (2) (1999) 177–228.
[35] U. Srivastava, K. Munagala, J. Widom, R. Motwani, Query optimization over web services, in: VLDB, 2006, pp. 355–366.

Georgia Kougka is a Ph.D. candidate in the Informatics department of Aristotle University of Thessaloniki since 2012. Her Ph.D. project is funded by the Thales Research Funding Program of the Hellenic General Secretariat for Research and Technology and her research interests lie in the field of data flow optimization, flow task allocation and multi-objective optimization. More details can be found at http://delab.csd.auth.gr/~georkoug/.

Anastasios Gounaris is an assistant professor lecturer at the Dept. of Informatics of the Aristotle University of Thessaloniki, Greece. A. Gounaris received his Ph.D. from the University of Manchester (UK) in 2005. His research interests are in the area of distributed data management, resource scheduling, autonomic computing and adaptive query processing. More details can be found at http://delab.csd.auth.gr/~gounaris/.

Kostas Tsichlas is a lecturer in the Informatics Department of Aristotle University of Thessaloniki since 2008. He was awarded a Ph.D. diploma in 2004. His research interests include the design and analysis of algorithms and data structures and complexity with a focus on lower bounds as well as on Natural Algorithms. More details can be found at http://delab.csd.auth.gr/~tsichlas/.

J Data Semant (2012) 1:75–97
DOI 10.1007/s13740-012-0006-9

ORIGINAL ARTICLE

Metrics for the Prediction of Evolution Impact in ETL Ecosystems: A Case Study

George Papastefanatos · Panos Vassiliadis · Alkis Simitsis · Yannis Vassiliou

Received: 19 November 2011 / Revised: 30 April 2012 / Accepted: 16 May 2012 / Published online: 4 July 2012

Abstract The Extract-Transform-Load (ETL) flows are essential for the success of a data warehouse and the business intelligence and decision support mechanisms that are attached to it. During both the ETL design phase and the entire ETL lifecycle, the ETL architect needs to design and improve an ETL design in a way that satisfies both performance and correctness guarantees and often, she has to choose among various alternative designs. In this paper, we focus on ways to predict the maintenance effort of ETL workflows and we explore techniques for assessing the quality of ETL designs under the prism of evolution. We focus on a set of graph-theoretic metrics for the prediction of evolution impact and we investigate their fit into real-world ETL scenarios. We present our experimental findings and describe the lessons we learned working on real-world cases.

Work conducted in the context of the "EICOS: foundations for personalized Cooperative Information Ecosystems" project of the "Thales" Programme. The only source of funding for this research comes from the European Social Fund (ESF)-European Union (EU) and National Resources of the Greek State under the Operational Programme "Education and Lifelong Learning (EdLL)".

G. Papastefanatos
Institute for the Management of Information Systems, Athens, Greece
e-mail: gpapas@imis.athena-innovation.gr

P. Vassiliadis
University of Ioannina, Ioannina, Greece
e-mail: pvassil@cs.uoi.gr

A. Simitsis (✉)
HP Labs, Palo Alto, CA, USA
e-mail: alkis@hp.com

Y. Vassiliou
National Technical University of Athens, Athens, Greece
e-mail: yv@cs.ntua.gr

Keywords Database evolution · Database metrics · Evolution case study

1 Introduction

Having accurate and up-to-date data warehouses is essential for Business Intelligence and Decision Support. A data warehouses design, apart from performance guarantees, should also provide correctness guarantees. Every time an evolution event occurs anywhere in the warehouse environment (e.g., a design change at the operational sources) it should be smoothly absorbed without causing any further inconvenience. For achieving this, the warehouse and its counterparts should be easily maintainable and the process of populating it should not be destructed by evolution events.

The Extract-Transform-Load (ETL) flows constitute the backbone of a typical data warehouse architecture. Most of the research for improving ETL designs has focused solely on improving performance. However, based on practical experience, maintenance makes up for up to 60 % of the resources spent in a warehouse project [34], and therefore, maintainability is an important factor for the determination of the quality of a design [19,36]. Although practitioners are well aware of this problem, still, we miss a formal and concrete answer to fundamental questions like "How good is an ETL design?" and "What makes an ETL design good or bad?". Typically, such questions are answered by a set of empirical rules based on practical observations of the past, as well as rules of thumb that have been established by expert practitioners and despite their value, they simply transfer the lessons learned the hard way in the "craft" of ETL design. Most of these rules consider only structural properties of the ETL flow or constructs internal to the underlying databases and do not take into account neither the incorporation of constructs surrounding the databases, nor the fact that a software construct, and especially an information system, evolves over time.

In practice, the problem is hard since changes in the schema of database-centric systems affect not only both its internals but also the surrounding deployed applications. Hence, the minimal interdependence of these software modules results in higher tolerance to subsequent changes and should be measured with a principled theory. Related work for evolution data-intensive applications [9], view redefinition [10,17,26], and data warehouse evolution [3,5,11,14] has provided rewriting techniques and theoretical cost models. Yet, a well-founded model, specifically tailored for the graph-based nature of ETL flows that assesses their vulnerabilities to changes is missing.

Related work also includes an approach to impact analysis and management of schema evolution, which represents the structural properties of the data warehouse schema, along with any views and queries defined over this schema, as a graph [30]. Our graph-based model captures all the parts (or, modules) of an environment, i.e., relations, views, and queries (which are practically the parts of ETL scripts that work the underlying data, or the elementary activities of a GUI-based scenario that are involved in the ETL process). Then, edges correspond to part-of or provider–consumer relationships. Given a database configuration, the impact of a schema change on the rest of the system is determined by exploiting the structure of the graph (i.e., by propagating the impact of the change via the involved edges). This clearly relates the structure of the graph, and its edges in particular, with the possibility that a component of the environment (a node in the graph) is affected by a certain evolution event. Furthermore, the evolution of the entire environment is regulated with the use of certain policies applied to the graph constructs. Example policies include either propagate/block a change or prompt the user for action. This way, administrators can regulate the evolution management, in a semi-automatic way, when changes on the database schema occur. Another research work employs a set of graph-theoretic metrics to measure evolution impact in data warehouse environments [29]. Although informally and briefly introduced in that work, these metrics are either degree-related or entropy-based metrics and compute the degree of dependence of nodes based on the structural properties of the system design from both a graph theoretic and an information theoretic perspective. However, these two works have not been adequately tested in real-world, large-scale applications.

In this paper, we built upon the aforementioned approaches with the goal of validating and experimentally assessing the proposed methods and metrics in real-world settings. We formally present these metrics and show that such metrics typically act as predictors for the vulnerability of a software module (either internal like a relation or external like a query) in a database-centric environment to future changes to the structure of the environment. Thus, we answer the aforementioned questions on the design quality of an ETL scenario from the perspective of maintenance.

Our experimental evaluation has been performed with a home-grown, publicly available, software tool, namely Hecataeus, which allows us to monitor evolution and perform evolution scenarios in database-centric environments. (For implementation details, the interested reader could read our ICDE'10 demo paper, Papastefanatos et al. [31].) The experimental analysis is based on a 6-month monitoring of seven real-world ETL scenarios processing data from statistical surveys. Our main goal was to examine different metrics over various ETL configurations and evolution events for assessing the usefulness and applicability of the proposed metrics (e.g., how well do they actually predict the impact of evolution events on a design construct). An additional desired objective was to identify which metric works best in different ETL configurations. Based on our findings, observations, and analysis, we disclose a list of lessons learned through this multi-month work.

In a nutshell, we have identified the schema size and module complexity as two important factors for the vulnerability of a system.

Schema sizes. The size of the schemas involved in an ETL design significantly affects the design vulnerability to evolution events. For example, source or intermediate tables with many attributes are more vulnerable to changes at the attribute level. Thus, a good design may involve tables with smaller schemas (e.g., we should maintain intermediate tables with a small number of attributes).

Functionality of ETL activity. The internal structure of an activity plays a significant role for the impact of evolution events on it. For example, activities with high out-degree and out-strengths tend to be more vulnerable to evolution and activities performing an attribute reduction (e.g., through either a group-by or a projection operation) are in general, less vulnerable to evolution events.

Module-level design. The module-level design of an ETL flow also affects the overall evolution impact on the flow. For example, it might be worthy to place schema reduction activities early in an ETL flow to restrain the flooding of evolution events. However, as we discuss in Sect. 5, such heuristics that significantly improve maintainability of ETL flows might contradict the normal practice for improving ETL performance.

In addition, we tested our metric suite against various ETL designs and have identified what metric provides better evolution prediction for specific ETL constructs. For modules with a single provider, the out-degree and out strength metrics (described in Sect. 3), which capture the dependencies with an adjacent module, provide better results. However, transitive degree metrics may act as predictors for the evolution of a module when it has many different providers and paths to evolving sources (e.g., queries).

Retrospectively, we can report that experimenting with real-world evolution scenarios in data-centric environments—and especially in ETL and data warehousing—is a difficult, long-termed, and time-consuming process. Such a process comprises a series of tasks, responsible for (a) recording all metadata information and workload definitions; (b) modeling and analyzing the dependencies between them; (c) collecting and categorizing all different types of evolution changes that occurred at different time periods and on different parts of the environment, and (d) recording how often each part of the environment (e.g., a query, a view) is affected by each change. Besides the non-trivial technical difficulties and effort needed for completing these tasks, in some cases, like for tasks (a) and (c), we had to deal with political and organizational issues as well. That is because more than one team of an organization is typically involved in these tasks and getting permissions, engaging people in exchanging and sharing information, and depending on other people's availability are not easy to carry tasks. This is an additional reason in favor of having a system toward the automatic or semi-automatic handling of evolution events in ETL and in general, in information management environments.

Outline The rest of this paper is structured as follows. Section 2 describes a graph-theoretic model for representing the constructs of a data warehouse environment. Section 3 presents the set of metrics. Section 4 presents our experimental findings. Section 5 provides a list of lessons learned. Finally, Sects. 6 and 7 discuss related work and conclude the paper, respectively.

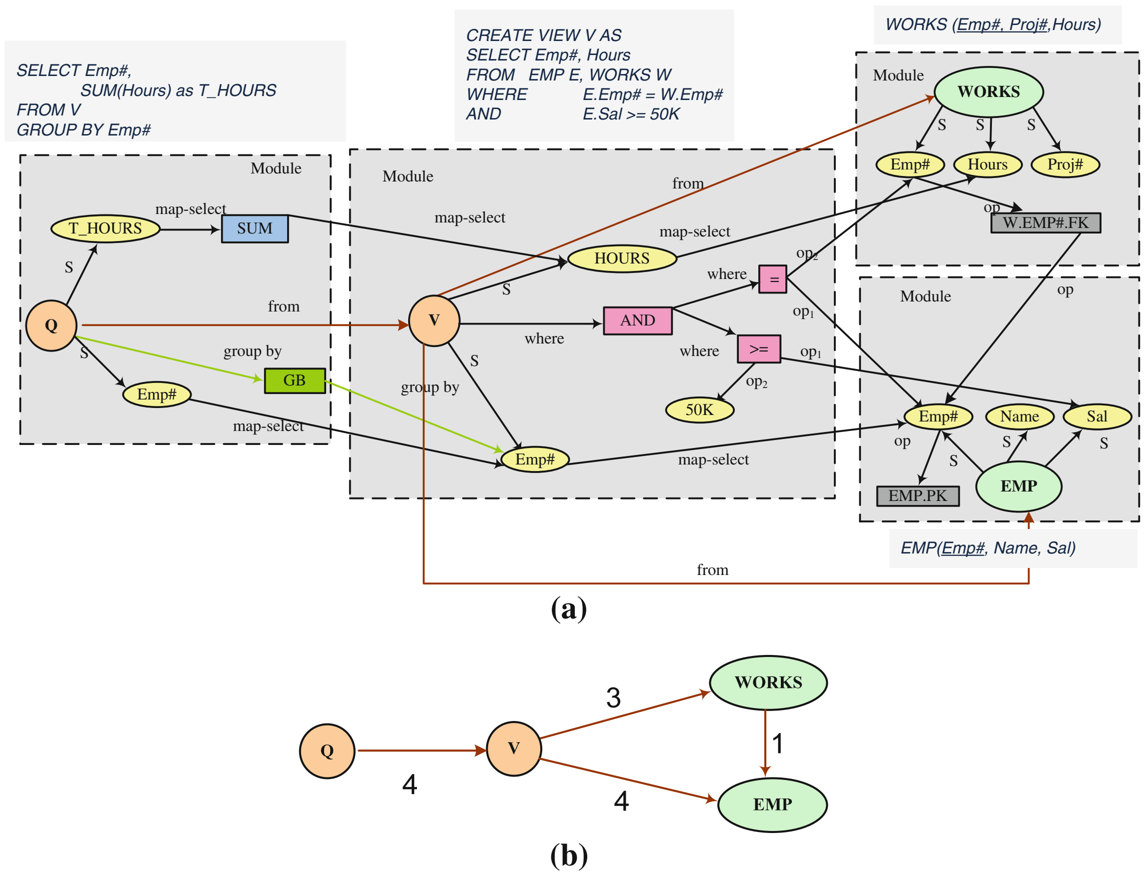
2 Modeling ETL Designs

In this section, we describe a graph-based model for ETL design. ETL designs are typically represented as graphs connecting activities and data stores. There are different styles for populating a data warehouse, like ETL, ELT, ETLT, and so on. Although these techniques have different performance characteristics, they do not differ in terms of modeling and thus, hereafter, we use the term ETL to capture all flavors of data warehouse population.

Our model uniformly covers relational tables, views, ETL activities, database constraints, and SQL queries as first class citizens. This model represents all such database constructs as a directed graph, named *evolution graph*, $G = (V, E)$. The nodes represent the entities of our model and the edges represent the relationships among these entities (mainly referring to part-of or provider-consumer relationships). The rationale for this modeling is to be able to represent *data-centric ecosystems* in a uniform way. In other words, we aim at a single, uniform way to model both database internals (like relations, views and constraints) and software modules external to the database (reports, forms, application programs, and so on). ETL flows offer a tight coupling of the database internal and external parts, along with the tight control of the application code by a small group of developers. Here, we present a brief description of our model. The interested reader may find a detailed model definition in another research paper [30].

A relation R $(\Omega_1, \Omega_2, \ldots, \Omega_n)$ in the database schema is represented as a directed graph, which comprises (a) a *relation node,* R, representing the relation schema; (b) n *attribute nodes*, $\Omega_1, \ldots, \Omega_n$, one for each of the attributes; and (c) n *schema relationships*, directing from the relation node towards the attribute nodes, indicating that the attribute belongs to the relation.

The graph representation of a Select-Project-Join-Group By (SPJG) query involves a new node representing the query, named *query node*, and *attribute nodes* corresponding to the schema of the query. The query graph is a directed graph connecting the query node with all its schema attributes, via *schema relationships*. In order to represent the relationship between the query graph and the underlying relations, we resolve the query into its essential parts: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`, each of which is eventually mapped to a subgraph. The edges connected the involved attribute and operand nodes are annotated as *map-select, from, and where relationships.* Aliases in the `FROM` clause (mostly needed in self-joins for our modeling) are annotated with *alias* edges. The direction of the edges is from the query node to the attribute nodes. `WHERE` and `HAVING` clauses are modeled via a left-deep tree of logical operands to represent the selection formulae; all the involved edges are annotated as *where* and *having relationships*, respectively. Nested queries are part of this modeling, too. For the representation of aggregate queries, we employ two special purpose nodes: (a) a new node denoted as GB, to capture the set of attributes acting as the aggregators; and (b) one node per aggregate function labeled with the name of the employed aggregate function, e.g., `COUNT`, `SUM`, `MIN`. For the aggregators, we use edges directing from the query node towards the GB node that are labeled `<group-by>`, indicating *group-by relationships*. Then, the `GB` node is connected with each of the aggregators through an edge tagged also as `<group-by>`, directing from the `GB` node towards the respective attributes. These edges are additionally tagged according to the order of the aggregators; we use an identifier i to represent the ith aggregator. Moreover, for every aggregated attribute in the query schema, there exists an edge directing from this attribute towards the aggregate function node as well as an edge from the function node towards the respective relation attribute. Both edges are labeled `<map-select>` indicating the mapping of the query attribute to the corresponding relation attribute through the aggregate function node. The



```
SELECT Emp#,
        SUM(Hours) as T_HOURS
FROM V
GROUP BY Emp#
```

```
CREATE VIEW V AS
SELECT Emp#, Hours
FROM   EMP E, WORKS W
WHERE        E.Emp# = W.Emp#
AND          E.Sal >= 50K
```

```
WORKS (Emp#, Proj#,Hours)
```

```
EMP(Emp#, Name, Sal)
```

Fig. 1 **a** Graph and **b** abstract representation of an aggregate query on top of a view defined over two relations

representation of the `ORDER BY` clause of the query is performed similarly.

Functions used in queries are denoted as a special purpose node `F` having the name of the function. Each function has an input parameter list comprising attributes, constants, expressions, and nested functions, and one (or more) output parameter(s). SQL Views are considered either as queries or relations (materialized views). Finally, DML and loading statements are modeled as simple SQL queries.

Figure 1a depicts the proposed graph representation for two relations, `EMP` and `WORKS`. `EMP` has three attributes, `Emp#`, which is the primary key, `Name` and `Sal`. `WORKS` relation comprises `Emp#` (foreign key to EMP primary key), `Proj#` and `Hours` attributes. On top of these relations, there is a view performing a join operation and filtering the employees having `SAL` more than 50 K. Finally, the graph depicts an aggregate query defined on top of this view.

Moreover, an ETL activity (e.g., a loading, cleansing, filtering operation, etc.) is modeled as an SQL view defined over the sources of the activity; furthermore, an ETL workflow is modeled as a sequence of views corresponding to the activities of the flow.

A module is a sub-graph of the overall graph in one of the following patterns: (a) a relation with its attributes and all its constraints, (b) a view with its attributes, functions and operands, and (c) a query with all its attributes, functions and operands. Modules are disjoint with each other and connected through edges concerning foreign keys, map-select and so on. Within a module, we distinguish *top-level* and *low-level* nodes. Top level nodes are used to signify the identity of the module; for that purpose, query, relation and view nodes are used as top-level nodes. Low-level nodes comprise the rest of the module. Edges are classified into *provider* and *part-of* relationships. Provider edges are inter-module relationships, whereas part-of edges are intramodule relationships In Fig. 1, the graph comprises four modules corresponding to the query, view and the relation subgraphs.

Zoomed out graph The graph that we have presented so far has the benefit of accurately representing the interrelationships of the involved constructs at the finest level of detail (practically the attribute level when data-centric ecosystems are involved). As usually happens, this comes at a price: the graph soon becomes large and crowded with all the details of internal representation of the modules. In order to (a) concisely represent the overall graph in main memory and (b) visually depict it, whenever the scale becomes too large, it is useful to zoom out the graph into a summary of appropriate structure and size. The zoomed out graph is an abstraction of the detailed evolution graph which comprises only top level nodes and edges between them. Abstracting the graph into a modular representation at a coarser level of detail involves the following steps: (a) for each query, view or relation module, all low-level nodes and intramodule edges are suppressed and only the respective top-level node is retained, and, (b) all inter-module edges apart from *from* and *foreign key* edges are dropped. A surviving edge between two modules is annotated with a weight corresponding to the number of the edges that originally connected the two modules. We call this weight the *strength* of the edge as it assesses how tightly the involved modules are coupled. Figure 1b depicts the abstract modular representation of Fig. 1a.

Events The space of potential events comprises the Cartesian product of two subspaces; specifically, (a) the space of hypothetical actions (addition/deletion/modification), and (b) the space of the graph constructs sustaining evolution changes. We consider and collect several cases of data warehouse evolution events, such as a dimension is removed, or renamed, the structure of a dimension table is updated, (e.g., addition, removal or modification of a dimension attribute), a fact table is completely decoupled from a dimension (deletion of a FK) or decoupled from one dimension and coupled to another (update of a FK), the measures of a fact table change, or the source table of an ETL is altered. To avoid overloading the text, we refer the interested reader to ([30], section 3.1) for a detailed description of events.

An update can signify a change of data types or a renaming of a construct; our practical experience indicates that it mostly refers to the latter. We do not check for additions of fact, dimension, or source tables, because such events do not result in a direct impact on any other logical warehouse construct per se. Given these changes that can occur to a data warehouse, their basic impact is that all software modules that use these database structures must be rewritten. The impact can be both syntactic (in the sense that all views and queries using a deleted attribute will crash) and semantic (in the sense that a new attribute in a relation or a modified condition in a view might require a rewriting of all the queries that use it). Assume for example that an attribute *FullName* is split to attributes *FirstName* and *LastName* or a view condition '*Year* = 2007' is altered to '*Year* > 2006'. The former change has syntactic impacts on all the queries using the attribute and the latter has semantic impact, since some of the queries using the view require exactly values of 2007, whereas some others will serve the purpose with any value greater than 2006.

Handling of events Given an event posed to one of the warehouse constructs (or, equivalently, to one of the nodes of the graph of the warehouse that we have introduced), the impact involves the possible rewriting of the constructs that depend upon the affected construct either directly, or transitively. In a non-automated way, the administrator has to check all of these constructs and restructure the ones he finds appropriate. This process can be semi-automated using our graph-based modeling and annotating the nodes and the edges of the graph appropriately with policies in the event of change. Assume for example, that the administrator guarantees to an application developer that a view with the sum of sales for the last year will always be given. Even if the structure of the view changes, the queries over this view should remain unaffected to the extent that its `SELECT` clause does not change. On the contrary, if a query depends upon a view with semantics '*Year* = 2007' and the view is altered to '*Year* > 2006', then the query must be rewritten.

The main idea in our approach involves annotating the graph constructs (relations, attributes, and conditions) sustaining evolution changes (addition, deletion, and modification) with policies that dictate the way they will regulate the change. Three kinds of policies are defined: (a) *propagate* the change, meaning that the graph must be reshaped to adjust to the new semantics incurred by the event; (b) *block* the change, meaning that we want to retain the old semantics of the graph and the hypothetical event must be vetoed or, at least, constrained, through some rewriting that preserves the old semantics; and (c) *prompt* the administrator to interactively decide what will eventually happen. Papastefanatos et al. [28] have proposed a language that greatly alleviates the designer from annotating each node separately and allows the specification of default behaviors at different levels of granularity with overriding priorities.

Given the annotation of the graph, there is also a simple mechanism that (a) determines the status of a potentially affected node on the basis of its policy, (b) depending on the node's status, the node's neighbors are appropriately notified for the event. Thus, the event is propagated throughout the entire graph and affected nodes are notified appropriately. The `STATUS` values characterize whether (a) a node or one of its children (for the case of top-level nodes) is going to be deleted or added (e.g., `TO-BE-DELETED, CHILD-TO-BE-ADDED)` or (b) the semantics of a view have changed, or (c) whether a node blocks the further propagation of the event (e.g., `ADDITION-BLOCKED)`.

3 Metric Suite

This section presents a set of metrics based on graph theoretic properties of the evolution graph for measuring and evaluating the design quality of a database centric environment with respect to its ability to sustain changes. For our analysis, we examine the graph (a) at its most detailed level (node level) that involves all the attributes of relations, views

Table 1 Degree related metrics

Notation	Metrics for any node
$D^I(v)$	In-degree of a node v
$D^O(v)$	Out-degree of a node v
$D(v)$	Degree of a node v
$TD^I(v)$	In-transitive degree of a node v
$TD^O(v)$	Out-transitive degree of a node v
$TD(v)$	Transitive degree of a node v
$D^{Is}(v)$	In-degree of a module v
$D^{Os}(v)$	Out-degree of a module v
$D^s(v)$	Degree of a module v
$TD^{Is}(v)$	In-transitive degree of a module v
$TD^{Os}(v)$	Out-transitive degree of a module v
$TD^s(v)$	Transitive degree of a module v

Table 2 Entropy-based metrics

Notation	Metric
$H(v)$	Entropy of a node v
$H^s(v)$	Entropy of a module v

and queries, along with the internals of the queries, and, (b) at a coarse level of abstraction (module level), where only relations, views and queries are present. An earlier work, has briefly introduced these metrics [28]. Here, we formally define them and provide the intuition and a detailed definition for each metric. The whole set of proposed metrics is presented in Tables 1 and 2.

3.1 Degree-Related Metrics

The first family of metrics concerns simple properties of each node or module in the graph and specifically the degree of nodes. The main idea lies in the understanding that the in-degree, out-degree and total degree of a node v demonstrate in absolute numbers the extent to which (a) other nodes depend upon v, (b) v depends on other nodes, and (c) v is interacting with other nodes in the graph, respectively.

Specifically, let $G(V, E)$ be the evolution graph of a database centric environment and $v \in V$ a node of the graph; then

Definition 1 Degree of Node: The *In-degree*, D^I(v), *Out-degree*, D^O(v) and *Degree, D*(v) of the node v are the total number of incoming, outgoing and adjacent edges to v. That is

$$D^I(v) = |e_{\text{in}}|, \text{ for all edges } e_{\text{in}} \in E \text{ of the form } (y_i, v), y_i, v \in V$$

$$D^O(v) = |e_{\text{out}}|, \text{ for all edges } e_{\text{out}} \in E \text{ of the form } (v, y_i), y_i, v \in V$$

$$D(v) = D^I(v) + D^O(v)$$

Transitive Degrees. The simple degree metrics of a node v are good measures for finding the number of nodes that directly depend on v, or on which v directly depends on, but they cannot detect the transitive dependencies between nodes. This typically occurs whenever a query accesses a view, which is of course defined over one or more views and relations. The metrics related to simple degrees cannot capture the fact that a change in a relation can eventually propagate to a large number of dependent modules *transitively*. Take, for example, the case of an ETL flow where a source relation may feed only a single activity; however, a change in this relation can transitively propagate and affect the entire workflow. In the context of our graph model, we say that a node v_1 is transitive dependent on another node v_2 if there is a path from v_1 towards v_2. Therefore, we employ the following definition for the transitive degrees of a node v with respect to the rest of the graph:

Definition 2 Transitive Degree of Node: The *In-Transitive, TD*$^{\text{I}}$(v), *Out-Transitive*, *TD*$^{\text{O}}$(v), and *Transitive degree, TD*(v) of a node $v \in V$ with respect to all nodes $y_i \in V$ are given by the following formulae:

$$TD^I(v) = \sum_{y_i \in V} |paths(y_i, v)|, \ y_i \in V$$

$$TD^O(v) = \sum_{y_i \in V} |paths(v, y_i)|, \ y_i \in V$$

$$TD(v) = TD^I(v) + TD^O(v)$$

Module degree. The aforementioned metrics are able to capture the significance of individual nodes of the graph at a fine-grained level. However, it is quite possible that administrators and designers are interested to see the graph properties at the module level. A first possible reason for this requirement is the graph's size: the administrators/designers might be willing to pay a small price in accuracy in favor of faster computation. Also, the metric properties of the modules (seen as black boxes) per se could be of interest to the administrators and the developers. To address this requirement, one can measure the degrees of *the zoomed-out* graph. As already mentioned in chapter 2, zooming-out operation on the graph provides an abstract view of the modules of the graph, which comprises only top-level nodes, i.e., relations R, views *VS* and queries Q and edges between them. All edges are annotated with a strength corresponding to the number of edges previously connecting these modules. Thus, we define the module degree for a node of the zoomed out graph as

Definition 3 Degree of Module (Strength): The *In-Module*, $D^{Is}(v)$, *Out-Module*, $D^{Os}(v)$ and *Module Degree, D*$^s(v)$of a node v are given by the following formulae:

$$D^{Is}(v) = \sum_{y_i \in V} strength(y_i, v), \ y_i, v \in \{R, Q, VS\}$$

$$D^{Os}(v) = \sum_{y_i \in V} strength(v, y_i), \; y_i, v \in \{R, Q, VS\}$$

$$D^{s}(v) = D^{Is}(v) + D^{Os}(v)$$

Module transitive degree. Similarly to above, we may extend the transitive degrees to the zoomed-out graph according to the following definition:

Definition 4 Transitive Degree of Module (Strength): The *In-Module, $TD^{Is}(v)$, Out- Module*, $TD^{Os}(v)$, and *Module Transitive degree, $TD^{s}(v)$* of a node v *∈{R,Q,VS}* with respect to all nodes y_i *∈{R,Q,VS}* are given by the following formulae:

$$TD^{Is}(v) = \sum_{y_i \in V} \sum_{e_p \in paths(y_i, v)} strength(e_p), \; y_i, v \in \{R, Q, VS\}$$

$$TD^{Os}(v) = \sum_{y_i \in V} \sum_{e_p \in paths(v, y_i)} strength(e_p), \; y_i, v \in \{R, Q, VS\}$$

$$TD^{\mathrm{S}}(v) = TD^{\mathrm{Is}}(v) + TD^{\mathrm{Os}}(v)$$

3.2 Entropy-Based Metrics

The last family of metrics presented is related to the information theoretic notion of entropy. Entropy is viewed as an arcane subject related somehow to uncertainty and information [32]. Given a set of events $A = [A_1, \ldots, A_n]$ with probability distribution $P = \{p_1, \ldots, p_q\}$, respectively, entropy is defined as the average information obtained from a single sample from A:

$$H(A) = -\sum_{i=1}^{n} p_i \log_2 p_i .$$

Entropy is strongly related to the information that is "hidden" in a probabilistic model. For instance, in a uniform probabilistic model, all events are equally likely to occur and therefore the entropy of the model is maximum.

In our evolution context, the notion of entropy is used to evaluate the extent to which a node is *likely* to be affected by a *random* evolution event on the graph. Intuitively, this likelihood is strongly related to the number of other nodes in the graph on which this node depends (i.e., connected to) either directly or transitively. Nodes connected via multiple paths to many parts are more likely to be affected if a random event occurs on the graph. Thus, entropy measures either the a priori uncertainty of the impact of an event on a part of the graph or equivalently the a posteriori amount of information we get from the knowledge that a part of the graph has been affected by an event. The more unpredictable the impact of a schema change on a part (either a module or a specific node) of the graph is, the higher the entropy is that characterizes this impact. For example, consider the graph of Fig. 2, where a query Q_1 is defined on top of two views, V_1 and V_2, which both access a single relation R_1; a third view, V_3, is defined on top of a second relation, R_2. Q_1 depends on three out of five modules in the graph, i.e., V_1, V_2 and R_1, and thus, it has a high *potential* that it will be affected by a random event on the graph, in contrast with V_3, which is connected only to R_2 and affected by changes occurring only to this relation.

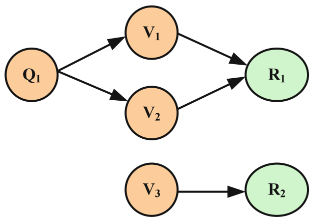


Fig. 2 A graph with a query, three views, and two relations

The following definitions introduce the metrics of the entropy of a node at the detailed and the zoomed out graph:

Definition 5 Entropy of Node: Assume a node v in our graph $G(V, E)$. We define the probability that $v \in V$ is affected by an arbitrary evolution event e over a node $y_k \in V$ as the number of paths from v towards y_k divided by the total paths from v towards all nodes in the graph, i.e.,

$$P(v|y_k) = \frac{|paths(v, y_k)|}{\sum_{y_i \in V} |paths(v, y_i)|}, \quad \text{for all nodes } y_i \in V.$$

Then, the information we gain when a node v is affected by an event that occurred on node y_k is $I(P(v|y_k)) = \log_2 \frac{1}{P(v|y_k)}$ and the entropy of node v with respect to the rest of the graph is then

$$H(v) = -\sum_{y_i \in V} P(v|y_i) \log_2 P(v|y_i), \; \text{for all nodes } y_i \in V.$$

Observe that high entropy values correspond to these parts of the graph, that are dependent on many providers either directly or transitively, *capturing in a "smoother" way than the local or the transitive degrees the dependencies in the graph.*

Definition 6 Entropy of Module: Moreover, we can apply the previously used technique to the zoomed out-graph $G^s(V^s, E^s)$, by defining the probability of a node $v \in V^s$ to be affected by an evolution event over a node $y_k \in V^s$ as

$$P^s(v|y_k) = \frac{\sum_{e_p \in paths(v, y_k)} strength(e_p)}{\sum_{y_i \in V^s} \sum_{e_p \in paths(v, y_i)} strength(e_p)}, \quad \text{for all nodes } y_i \in V^s.$$

with $e_p \in E^s$ being the edges of all the paths of the zoomed out graph stemming from v towards y_k. Similarly, the entropy of node $v \in V^s$ is

$$H^s(v) = -\sum_{y_i \in V^s} P^s(v|y_i) \log_2 P^s(v|y_i) \quad \text{for all nodes } y_i \in V^s.$$

A summary of the proposed set of metrics is provided in Tables 1 and 2.

3.3 Rationale for Our Approach and Broader Perspective

Before describing how the aforementioned metrics have been applied to a real case study, we discuss the rationale for the choice of this metric suite.

We base our approach on the modeling power of the graph proposed. Our graph-based model is simple, comprehensive, rigorously defined, and it has a uniform treatment of all involved constructs. Nodes correspond to both detailed entities like attributes or values and to higher level entities like relations or queries (this can further be abstracted to databases, scripts, libraries, and so on). Edges denote *any* form of relationship, with two kinds of relationship being the most prominent ones: (a) part-of (between high-level modules and their constituents) and (b) provider-consumer in terms of data provision. Both kinds of edges capture the notion of *dependency*: an edge $e(v, u)$ from node v to node u, signifies that node v is potentially affected whenever u is affected too.

We believe that dependency is the cornerstone of maintenance in data-centric ecosystems. To this end, we devised two families of metrics based on mathematical fundamentals to quantify the dependency of a node. The first family of metrics has to do with graph-theoretic properties. In the context of our studies, we focused on the properties of *individual nodes*, and, in fact, we opted to constrain ourselves to simple metrics like degrees and we explore the main kinds of degrees. At the same time, we also explore two different modes of locality. First, we are interested in the local degree, as a simple measure of direct dependence between nodes. Second, we operate in a workflow-like environment where data are "copied" from one module to another for further processing and thus, we explore the idea of assessing transitive degrees as measures of the overall dependency of a node to other nodes. The second family of metrics serves an information-theoretic rationale: what if a random evolution event appears in the graph? How likely is a node v to be affected by it due to its dependency to other affected nodes? Thus, we follow a mathematically founded, information theoretic approach to capture the vulnerability of a node to a random evolution event.

In both families, we do distinguish between coarse-grained metrics at the module level versus detailed metrics at the full extent of the graph. As already mentioned, the coarse-grained summary of the graph was intended to alleviate the user from the information overload of all the miniscule details of module internals at the full extent of the graph. However, due to the size of the graph, it is possible that large ecosystems will have to be assessed at a coarser detail level for performance reasons; then, the open question is how well do coarse-level metrics approximate the detailed ones.

From a subjective viewpoint, without excluding the possibility of more sophisticated metrics, we support the idea of *simple* metrics. In this paper, we are exploring the problem from an empirical perspective and are primarily interested to see what simple metrics appear to work well in the case study we have encountered. We deem *simplicity* as an inherent good quality that makes concepts easily understandable and explainable to the involved stakeholders (let alone efficiently computed). But even under the prism of simplicity, we do not claim that our metric suite exhausts all possibilities, either with respect to their mathematical foundation or with respect to practical intuition (or any other rationale that can be used to build a set of metrics); on the contrary, we do anticipate that other metrics can possibly be devised to assess the vulnerability of a node to change.

4 A Real-World Case Study and Experimental Validation

The metrics presented in Sect. 3 express the degree of dependence or importance of a node in an objective and quantifiable way. Yet, this is a characterization of the structural properties of a node within the graph. How accurately can we use the metrics to *predict* the vulnerability of a node to change? Is it fair to say that the *more dependent* a node is, the *higher the probability* that it is affected by evolution changes? To address this issue, we have evaluated the proposed metrics with real-world ETL scenarios. The goal of our analysis is to evaluate the extent to which our metrics are good indicators for the prediction of the effect that evolution events have. A clear desideratum in this context is the determination of the most suitable metric for this prediction according to the different types of evolved constructs.

4.1 Experimental Setting

In our experiments, we have studied a data warehouse scenario, which involves real-world evolution scenarios of ETL workflows that were monitored for a period of six months. The environment involves a set of seven real ETL workflows extracted from a Greek public sector's data warehouse maintaining information for farming and agricultural statistics. The ETL flows extract information out of a set of seven source tables, namely S1 to S7 and 3 lookup tables, namely L1 to L3, and load it to seven target tables, namely T1 to T7, stored in the data warehouse. The seven scenarios comprise

Table 3 ETL scenarios configuratrion

ETL	# Activities	Sources	Tmp Tables	Targets
ETL 1	16	L1,L2,L3,S1,S4	T1_Tmp, T2_Tmp, T3_Tmp	T1, T2, T3
ETL 2	6	L1,S2	T1_Tmp, T3_Tmp	T3
ETL 3	6	L1,S3	T1_Tmp, T3_Tmp	T3
ETL 4	15	L1,S4	T1_Tmp, T3_Tmp, T4_Tmp	T3, T4
ETL 5	5	S5	T1_Tmp, T5_Tmp	T5
ETL 6	5	S6	T1_Tmp, T6_Tmp	T6
ETL 7	5	S7	T1_Tmp, T7_Tmp	T7
Total	58			

Table 4 Number of attributes in ETL source tables

Table	S1	S2	S3	S4	S5	S6	S7	L1	L2	L3
# Attributes	59	160	82	111	13	7	5	7	19	7

Table 5 Distribution of events at the ETL tables

Source	Change type	Occurrence	Affected ETL
L1	Add Attribute	1	ETL 1, 2, 3, 4
L1	Add Constraint	1	ETL 1, 2, 3, 4
L2	Add Attribute	3	ETL 1
L3	Add Attribute	1	ETL 1
S1	Add Attribute	14	ETL 1
S1	Drop Attribute	2	ETL 1
S1	Modify Attribute	3	ETL 1
S1	Rename Attribute	3	ETL 1
S1	Rename Table	1	ETL 1
S2	Add Attribute	15	ETL 2
S2	Drop Attribute	4	ETL 2
S2	Rename Attribute	121	ETL 2
S2	Rename Table	1	ETL 2
S3	Rename Attribute	80	ETL 3
S3	Rename Table	1	ETL 3
S4	Add Attribute	58	ETL 1, 4
S4	Drop Attribute	26	ETL 1, 4
S4	Modify Attribute	1	ETL 1, 4
S4	Rename Attribute	27	ETL 1, 4
S4	Rename Table	1	ETL 1, 4
S5	Modify Attribute	2	ETL 5
S5	Rename Table	1	ETL 6
S6	Rename Table	1	ETL 6
S7	Rename Attribute	5	ETL 7
S7	Rename Table	1	ETL 7
T1	Drop Attribute	1	ETL 1
T1	Modify Attribute	1	ETL 1
T1_tmp	Drop Attribute	1	ETL 1-7
T1_tmp	Modify Attribute	1	ETL 1-7
T2	Add Attribute	15	ETL 1
T2	Modify Attribute	2	ETL 1
T2_tmp	Add Attribute	15	ETL 1
T2_tmp	Modify Attribute	2	ETL 1
T5	Modify Attribute	2	ETL 5
T5_tmp	Modify Attribute	2	ETL 5
Total		416	

a total number of 58 activities extracting, filtering and loading data into the target tables. They, also, make use of seven temporary tables (each target table has a temporary replica) for keeping data in the data staging area, as shown in Table 3. The warehouse maintains statistical information collected from surveys, held once per year via questionnaires. The survey data are primarily stored in the S1–S7 source tables and are subsequently processed so that they can be integrated in the organization's warehouse and queried. Each survey varies its schema according to the different number and types of questions comprising the survey's questionnaire; however, there are several common elements in all surveys. Table S1 holds information about the metadata of the survey (e.g., the year held, the sample size, etc.), which are rarely altered or renamed and mostly complemented or specialized yearly by adding new attributes in the survey's metadata. All other source tables (S2–S7) retain the answers to the questionnaires, where most alterations occur every year. The size of the schema of each table, in terms of number of attributes, is shown in Table 4.

Our choice for experimenting with these scenarios was based on their evolution behavior that satisfied the following criteria: The first criterion was the ease of collecting real evolution events. As statistical surveys are held once a year, most source tables are suffering the majority of evolution events during a short peak period when surveys are designed and modeled in the database. This fact enabled us to collect and analyze most evolution events at once. The second criterion was the number and diversity of events. The scenarios examined exhibit a large number of evolution events covering a broad variety of alterations on the source tables (see Table 5). Finally, the third criterion was the diversity in the designs of the ETL flows. The chosen ETL scenarios enabled us to evaluate our metrics in simple (e.g., ETL5, ETL6, ETL7) as well as more complex (e.g., ETL1) flows.

All ETL scenarios were source coded as PL\SQL stored procedures in the data warehouse. First, we extracted embedded SQL code (e.g., cursor definitions, DML statements, SQL queries) from activity stored procedures. Table definitions (i.e., DDL statements) were extracted from the source and data warehouse dictionaries. Each activity was represented in our graph model as a view defined over the previous activities, and table definitions were represented as relation graphs.

We have used a homegrown software artifact, called Hecataeus[1] for the graph representation, the application of changes on the source tables, the evaluation of the metrics on the graph, and the identification of the impact of evolution events. Hecataeus is an open-source software tool for enabling impact prediction, what-if analysis, and regulation of relational database schema evolution. Under the hood, it supports the proposed graph-based modeling technique and represents database schemas and database constructs, like queries and views, as graphs. Our tool enables the user to create hypothetical evolution events and examine their impact over the overall graph before these are actually enforced on it. It also allows the definition of rules (i.e., policies) on nodes of the graph (in the form of annotations) that regulate the propagation of the evolution impact on specific parts. Most importantly, Hecataeus includes a metric suite for evaluating the impact of evolution events and detecting crucial and vulnerable parts of the system.

Our study is based on the evolution of the source tables and their accompanying ETL flows, which has happened in the *context of maintenance due to the change of requirements at the real world*. As already mentioned, source S1 stores the constant data of the surveys and did not change a lot. The rest of the source tables (S2–S7), on the other hand, sustained maintenance. The recorded changes in these tables mainly involve restructuring, additions, and renaming of the questions comprising each survey, which are furthermore captured as changes in the source attributes names and types. The set of evolution events includes renaming of relations and attributes, deletion of attributes, modification of their domain, and last, addition of primary key constraints. We have recorded a total number of 416 evolution events and the number of events per table is shown in Table 5. Observe that the majority of evolution changes concerns attribute renaming and attribute additions. These findings were due to the evolution context of the examined warehouse sources.

The last column in Table 4 shows the flows as affected by each change. L1 table is used in 4 ETL flows (1–4), S4 in 2 flows, namely ETL1 and ETL4, whereas all other source tables are used only to one flow. The most evolved table is S2 with a total of 141 changes and S4 follows with 113 changes. S2, however, supplies only one flow (ETL 2), whereas S4 supplies both ETL1 and ETL4. In addition, schema changes were applied in T_1, T_2, and T_5 target tables and their respective temporary tables as a result of the changes in the ETL sources. All evolution changes were applied in the form of annotations on the nodes of the graph.

Summarizing, the configuration of our experiments involved representing the ETL workflows in our graph model as well as the recorded evolution events on the nodes of the source, lookup and temporary tables. We then applied each event sequentially on the graph assuming that no rules constrain the propagation of the change towards the nodes of the graph. We, finally, monitored the impact of the change towards the rest of the graph by recording the times that a node has been affected by each change.

4.2 Experimental Validation

We have first evaluated the graph metrics on the seven ETL graphs and then applied the evolution events of Table 4 sequentially on these ETL graphs. We monitored each node of the graphs on how many times it was affected by an event. This measurement constitutes the baseline measurement that simulates what would actually happen in practice. This baseline measurement is compared with all measured metrics. In the rest, we discuss our findings organized according to each ETL flow.

ETL1. The first workflow (Fig. 3) comprises two source tables (S1, S4), three lookup tables (L1–L3), three target tables (T1–T3) along with their temporary tables and 16 activities. Both source tables contain a large number of attributes, namely 59 for S1 and 111 for S4 (there are no foreign keys defined), lookup tables are small in size, target tables T1 and T2 are two dimension tables with 74 and 38 attributes, respectively, whereas T3 is a fact table with 16 attributes. S1 data are extracted and loaded in the two dimension tables through the upper branch of the flow and to the fact table via the lower branch. S4 contains measure data that are loaded in the fact table. Regarding the functionality of the activities, the activities 1–5 perform extraction and filtering of data from the two source tables. Then, activity 9 joins the two sources and projects all attributes of S1 but only a small number of attributes of S4 (most data coming from S4 table are loaded via the ETL4 scenario). Activities 10–12 of the upper branch update the data with lookup values and activities Q2 and Q3 project and load data to T1, T2 temporary tables. In the lower branch, Q4 activity updates the data with values from L3 and loads it to T3 temporary table. Finally, activities 6–8 perform the final loading to the target tables of the data warehouse. Based on the functionality of each activity, we distinguish the *filtering* activities performing a select or a transforming operation on their source, (e.g., ETL1_ACT1, ETL1_ACT2, ETL1_ACT5, etc.), *joining* activities combining data from more than one sources (e.g., ETL1_ACT9, ETL1_ACT11,

[1] For more details about Hecataeus, we refer the interested reader to our ICDE'10 demonstration paper [31] and the project website: www.cs.uoi.gr/~pvassil/projects/hecataeus.

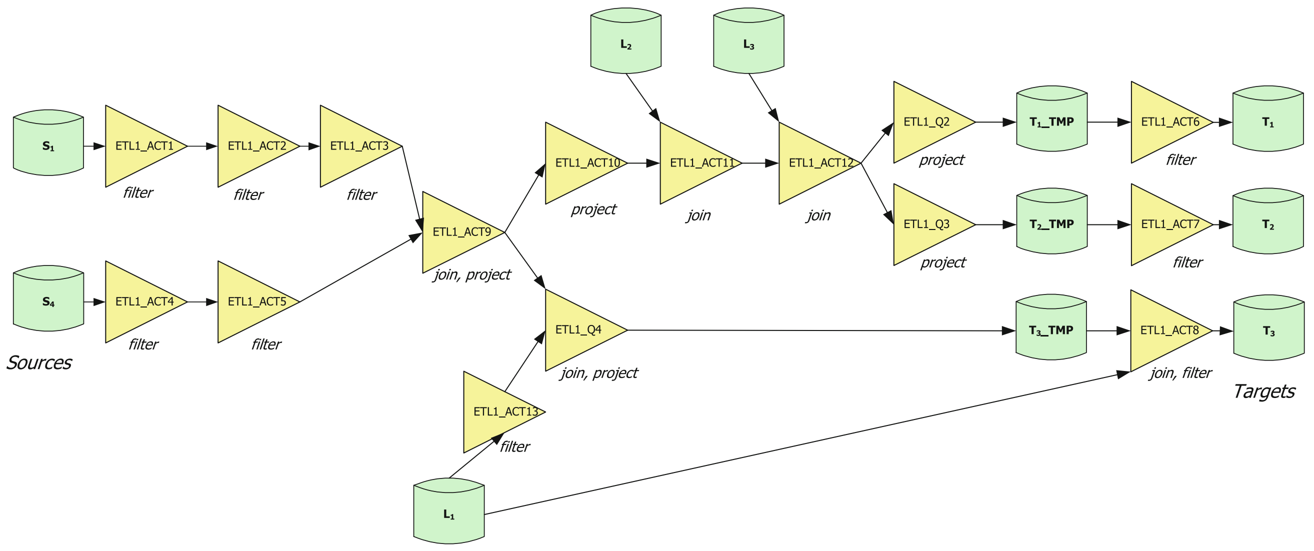


Fig. 3 Workflow of the first ETL scenario, ETL1

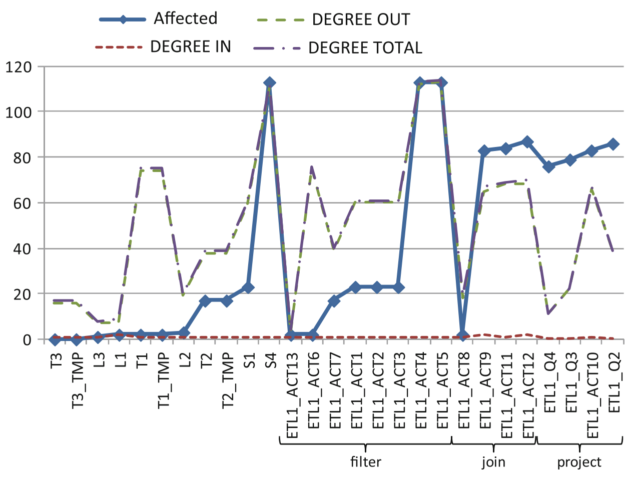


Fig. 4 Results for degree metrics for ETL1

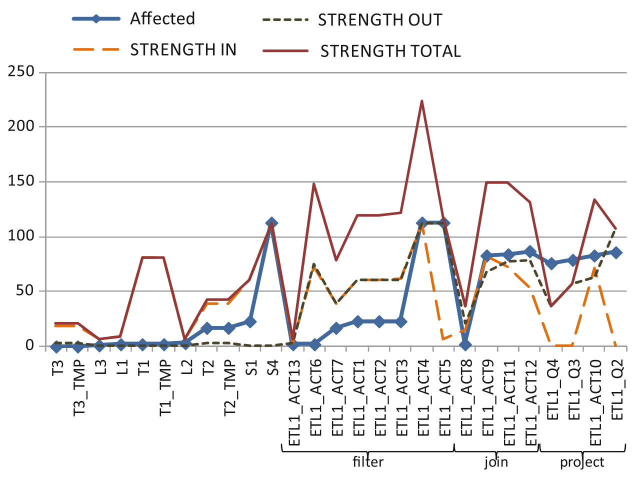


Fig. 5 Results for strength metrics for ETL1

ETL1_Q4, etc.) and finally activities that *project* a subset of the available attributes of their sources (e.g., ETL1_Q4, ETL1_ACT10, ETL1_ACT12, ETL1_Q2, ETL1_Q3, etc.).

In Figs. 4, 5, 6 and 7, we present the results for the first of the 7 ETL scenarios, grouped by the different types of metrics. In Fig. 4, we present the simple degree metrics, in Fig. 5 the transitive degree along with the entropy metrics (entropy and transitive metrics have been scaled up and down, respectively, for comparison reasons), and in Fig. 6 the strength metrics and last in Fig. 7 the transitive strength metrics. The goal is to show the overall trend of the examined metrics (i.e., we are not interested in the absolute numbers) with respect to the type of module and the times it is *affected* by all events that occurred at its source. In all figures, the tables are positioned on the left side followed by the activities. In Figs. 4 and 5 activities are first arranged by their type and then by the affected series, whereas in Figs. 6 and 7 by their type and then by their topological order in the workflow.

The *affected* series for the tables (in all figures) corresponds to the number of changes that occurred at their schemas as presented in Table 4. The most "evolved" table in ETL1 is S4, followed by S1, whereas T2 dimension table (along with the relevant temporary table) exhibits the highest number of changes among the data warehouse tables. This is due to the fact that most source schema changes, occurred at S1, have been exclusively propagated to the T2 dimension table, without altering T1 dimension table. Filtering activities are affected by all changes occurring at their source table. For example, ETL1_ACT1, ETL1_ACT2, and ETL1_ACT3 activities exhibit the same affected number with S1; ETL1_ACT4, ETL1_ACT5 with S4, etc. As we mentioned earlier, ETL_ACT9 projects all S1 attributes, but

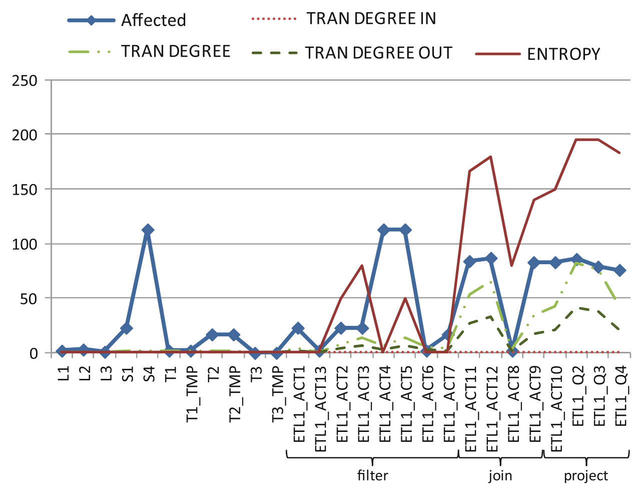


Fig. 6 Results for transitive degrees and entropy metrics for ETL1

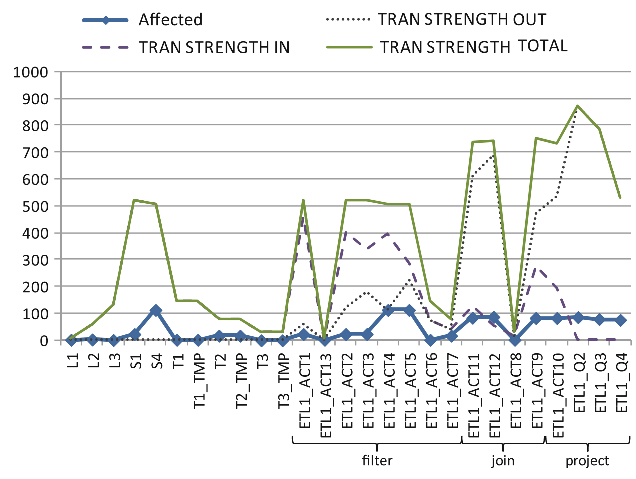


Fig. 7 Results for transitive strength metrics for ETL1

only a small number of S4 attributes. Thus, all other project and join activities, positioned after ETL_ACT9, are mostly affected by S1 evolution changes and S4 attribute additions. Next, we discuss our findings for each family of metrics.

The *in-degree* metric is significantly low and invariant to the number of events affected all modules. The *out-degree*, and consequently the total degree, follows proportionally the number of events that occurred on the source and lookup tables of the workflow, as well as on the T2 target table (the table that "absorbed" most source schema changes). The out-degree metric captures the size of the table schema, which seems to be a crucial factor for the evolution of the tables. The out-degree for all types of activities follows a similar trend. The out-degree for activities mostly captures the number of attributes that are projected by each activity and as a result, project activities exhibit low out-degree values. However, most activities have been affected by evolution proportionally to their out-degrees, except for some peaks such as ETL1_ACT6. ETL1_ACT6 activity depends on the T1_Temp table, which, however, has not sustained any schema changes.

The *strength* metrics for the tables (shown in Fig. 5) follows an opposite trend from the simple degrees. The *strength-out* is invariant to the affected series, whereas the *strength-in* and *total strength* follows the affected series for each table (with the exception of T1 and T1_temp). This can be explained by the fact that the strength-in metric for a table captures attribute dependencies between a module and this table. Figure 5 shows that the more "used" is a table, the higher the probability is to evolve. Filtering activities show a similar trend for both in and out strengths, except for ETL1_ACT5. The latter with all other activities show that the out-strength values follow more smoothly the affected series. Again, the peak for ETL1_ACT6 is due to its dependence from T1_Temp table.

The *transitive degrees and entropy* metrics, shown in Fig. 6, do not provide useful results for tables, because all values are insignificant to the affected series. This is not surprising as the entropy metric and transitive out-degree for a module captures the number of other modules on which this module depends transitively and tables have few or no dependencies on other modules. Regarding the activities, transitive out degree metrics and entropy follow smoothly the trend of the affected series, except for ETL1_ACT4 dive. ETL1_ACT4 exhibits a low value for the transitive degree metric, as it depends exclusively on S4, which, however, is affected by a large number of evolution events.

Similar to the simple strengths, the *transitive in* and *total strengths* follows the trend of affected series for tables. On the other hand, *transitive out* and *total strengths* seem to be more precise for activities, especially for join and filtering ones.

ETL2 and ETL3. The next two flows, ETL2 and ETL3, are shown in Figs. 8 and 10, respectively. Both flows behave similarly and load data from S2 and S3 to the T3 fact table; L1 and T1_temp tables are used as lookup tables. S2 contains 160 attributes and S3, 83 attributes. Both flows have no branches, whereas the activities mainly filter data from their sources and update lookup values. We observed that the number of events on these tables follows proportionally the tables' size and out-degree metric is again validated as a candidate predictor for the behavior of the evolution of the tables. The examined metrics on the activities of these two flows show similar results with ETL1. Out-Degree and out-strength are the most accurate predictors, but also transitive degree metrics (entropy and total transitive strength) follow the "affected" series. In Fig. 9a–d, we present the results for all the examined metrics for ETL2, and in Fig. 11a–d, the corresponding results for ETL3.

ETL4. Figure 12 shows the configuration of ETL4 and the respective results are shown in Fig. 13a–d. The ETL4 flow has one source, namely S4, comprising a fairly large number of attributes, 111. The two data warehouse tables, T3 and

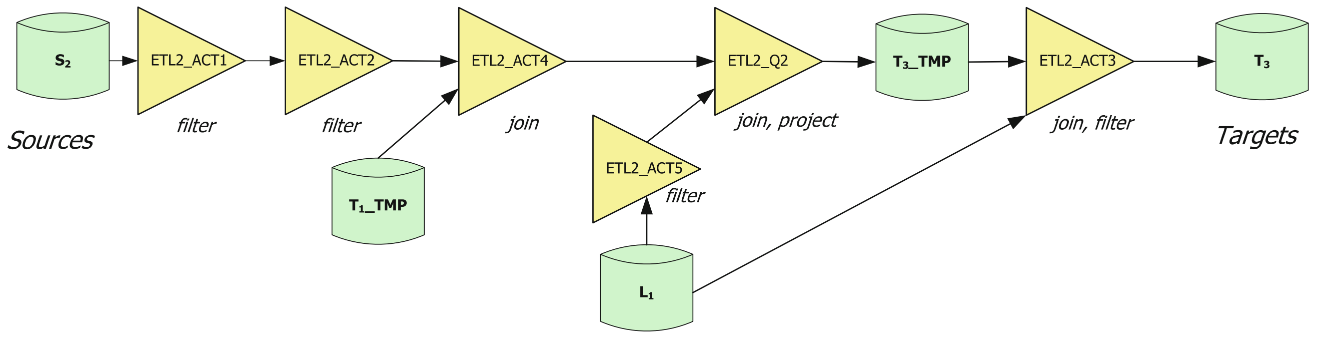


Fig. 8 ETL2 workflow

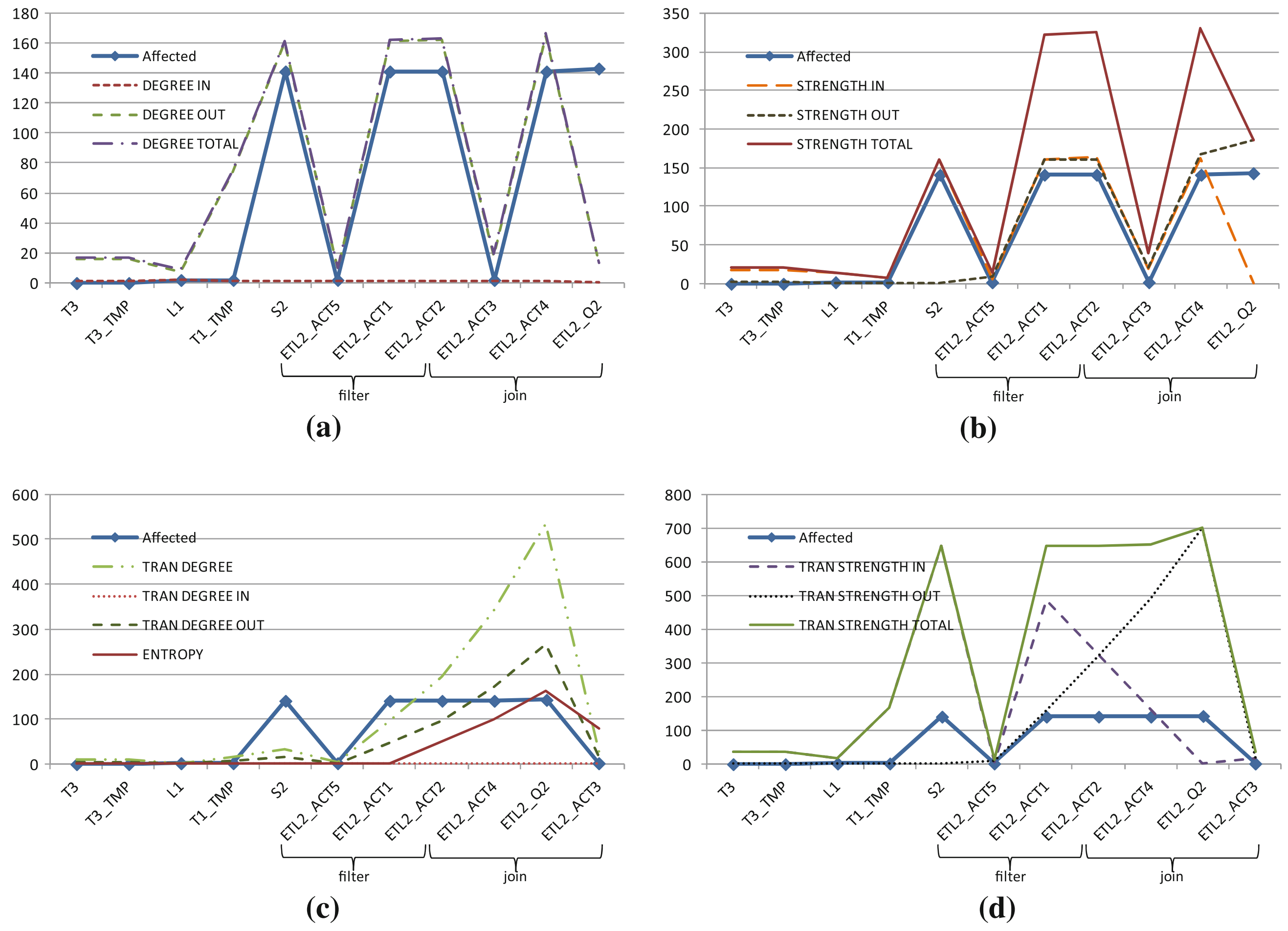


Fig. 9 Results for examined metrics for ETL2

T4, are both fact tables and are populated by a series of 9 activities, i.e., ETL4_Q2-ETL4_Q10. Each activity projects a different subset of source attributes and maps them to measure attributes in the data warehouse. Finally, L1 and T1 are used as lookup tables and ETL4_ACT3 and ETL4_ACT4 are used for transferring data from the temporary tables to the data warehouse tables.

The results for ETL4 are illustrated in Fig. 13a–d. Again, the out degree can be used as a predictor for the events that affect a table. Out-degree, total strength, and transitive total strength metrics are quite precise for the activities of this scenario. In contrast with the previous scenarios, scaled entropy and transitive out-degree are also proven to be good estimators for this setting. This can be explained by the fact that ETL4 is a short workflow, with only a few steps of processing and few transitive dependencies. Therefore, transitive degree metrics exhibit the same trend with the simple degree or strength metrics for all activities.

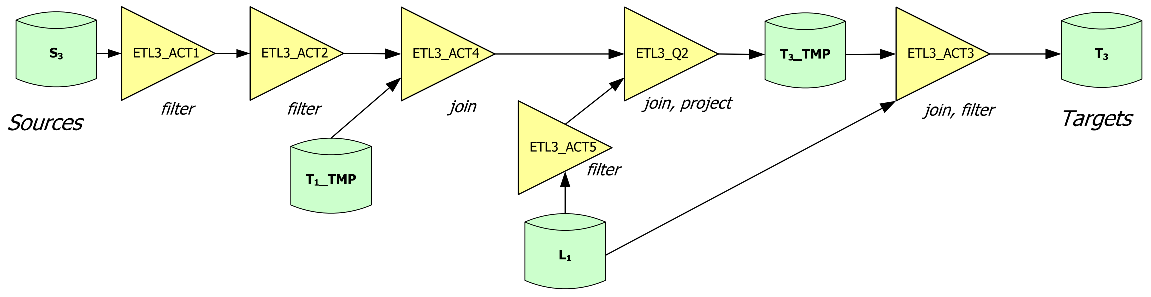


Fig. 10 ETL3 workflow

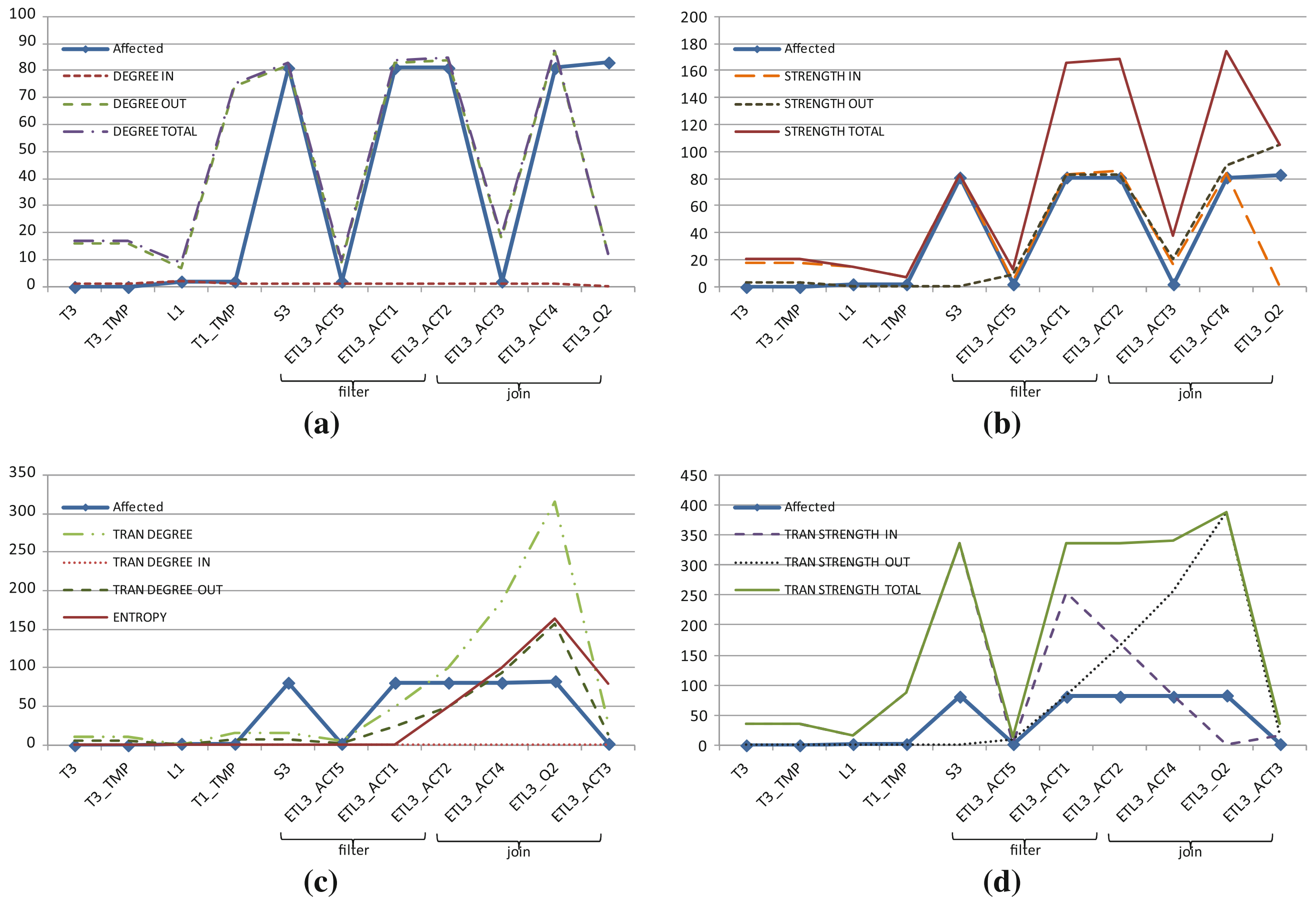


Fig. 11 Results for examined metrics for ETL3

ETL5, ETL6, and ETL7. The final three flows, ETL5, ETL6, and ETL7, are depicted in Fig. 14a, b, c, respectively. These three are similar to each other, loading data from three different source tables, namely S5 (with 13 attributes in its schema), S6 (with 7 attributes in its schema), S7 (with 4 attributes in its schema) to three target tables, T5, T6, and T7. T1_TMP table is used as a lookup table.

The results for these ETL flows are shown in Figs. 15, 16 and 17. The out-degree of relations follows proportionally the number of occurred events on them, except for S7, which unusually exhibits a high number of events with respect to its size. Activities in all three flows show similar behavior, where out degrees and strengths seem to provide more accurate results for the possible events on them (even though the sample of occurred events is low for these flows and affected series has very low values).

5 Lessons Learned

For several months, we have observed and experimented with genuine evolution events in real-world, ETL workflows. Our

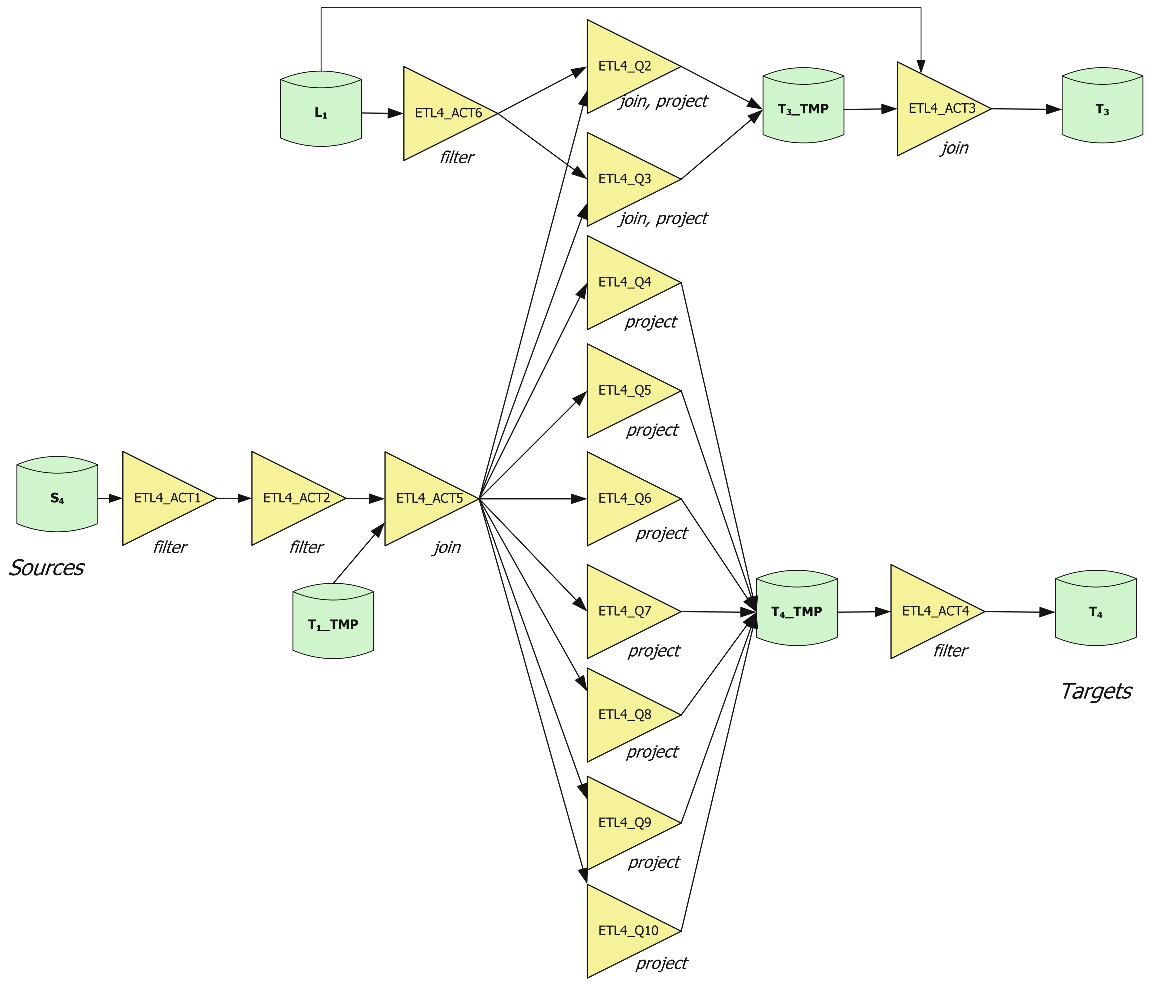


Fig. 12 ETL4 workflow

experimental findings are reported in Sect. 4. In the past, we did a similar exercise for a set of artificially created evolution scenarios [29]. Based on both activities, we assessed the metrics for predicting the behavior of a software system to evolution operations described in Sect. 3 and came up with some interesting insights regarding the value of our metrics. Next, we discuss our observations and suggest design strategies based on the results obtained by the use of these metrics.

Observations Our first observation, which is in accordance with intuition too, is that an important factor for the potential evolution of the whole or a part of a system is eventually its *schema size*. Especially in a workflow setting like the ETL environment, source or intermediate tables comprising many attributes in their schema are more likely to be altered and hence, more likely to affect the workflow they feed. Therefore, a particularly handy metric for the evaluation of the evolution potential of a workflow source table is practically the number of attributes it has (expressed by the *out-degree* metric in our setting). In terms of design, although it is often hard to change a source table, at least, the designer may choose to use intermediate tables with smaller schema sizes. For example, instead of just saving a snapshot of a table, she should try storing only its most valuable projection or instead of using a combination of production keys along with their origins, she should replace keys-origin pairs with surrogate keys.

Of course, in practice, looking just at a module size is not a panacea. There are counterexamples too, like in the case of the source table S7 in ETL7, where the number of evolution events is disproportional to the table size. Due to these cases, common practices like simple examination, through a set of standard queries, of the DB catalog tables do not suffice to get the most interesting metrics. Such cases are only identified with rigorous experimentation and for that, we need a well-structured set of metrics (like the ones presented in Sect. 3) to avoid dealing with exponentially perplexing situations as the project complexity increases.

Based on the results reported in Sect. 4, we observed that the most accurate and suitable metrics for all module types are the *out-degree* and *out-strength* metrics. On the one hand,

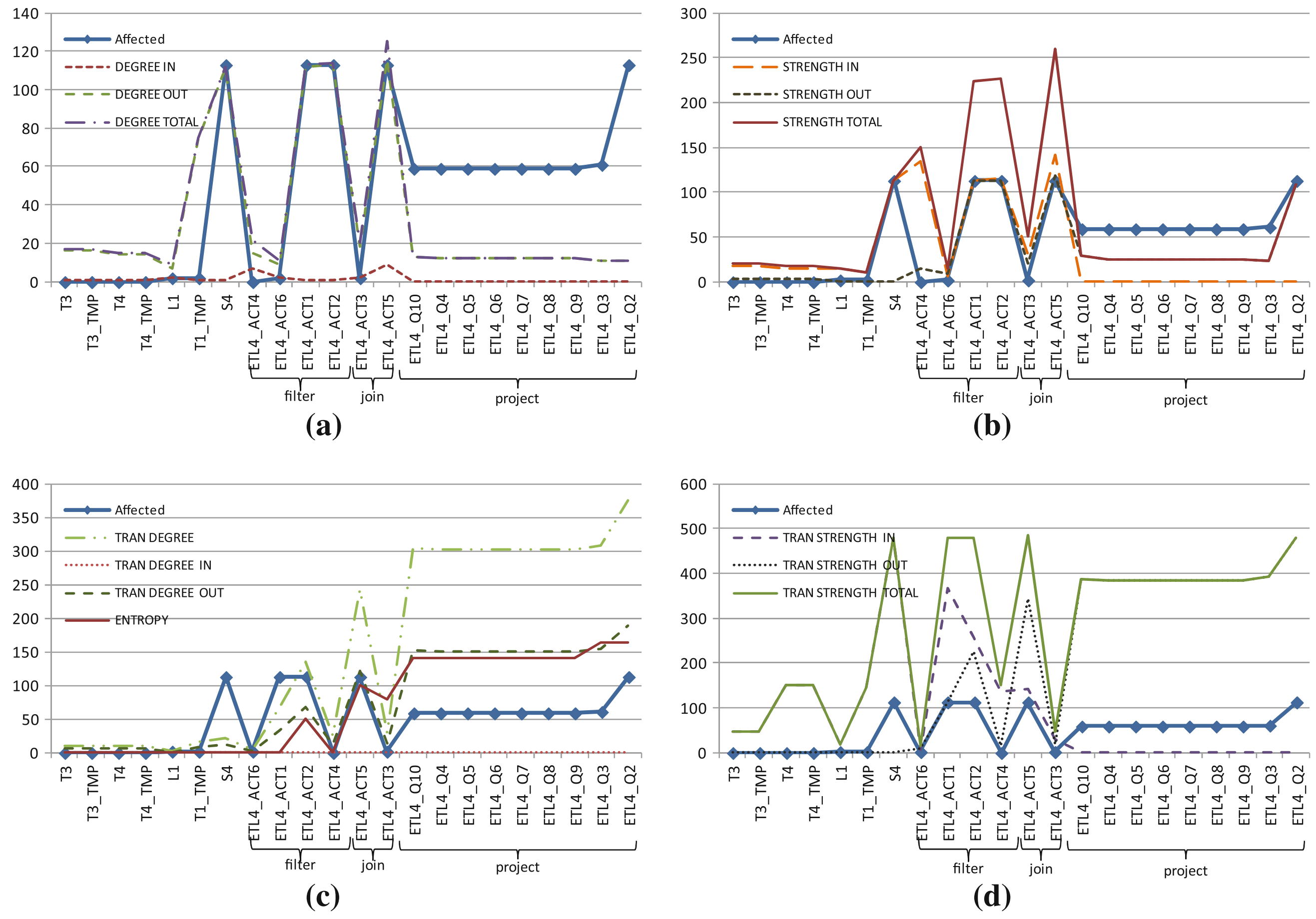


Fig. 13 Results for examined metrics for ETL4

when a module has only one provider, then it is safer to take into account the out-degree/out strength metrics, which capture the dependencies with an adjacent module. On the other hand, transitive degree metrics may act as predictors for the evolution of a module when it has many different providers and paths to evolving sources like for example Q2 and Q3 queries in ETL4 (see Fig. 12c, d) or ETL1_ACT10 - ETL1_ACT12 in ETL1 (see Figs. 6, 7).

Therefore, another observation is that the internal structure of each activity plays a significant role for the impact of evolution events on it. Activities with high *out-degree* and *out-strengths* tend to be more vulnerable to evolution. For example, such activities may project or use in conditions, a large number of attributes from their sources (either previous activities or tables). The out-degree captures the projected attributes by an activity, whereas the out-strength captures the total number of dependencies between an activity and its sources. Activities with joins between many sources tend to be more affected than activities sourced by only one provider, but still, the most decisive factor seems to be the activity size. Thus, activities that perform an attribute reduction on the workflow through either a group-by operation or a projection of a small number of attributes are in general, less vulnerable to evolution events and propagate the impact of evolution further away on the workflow (e.g., Q4 in ETL1 or Q2–Q10 in ETL4). In contrast, activities that perform join and selection operations on many sources and result in attribute preservation or generation on the workflow have a higher potential to be affected by evolution events (e.g., observe the activities ETL1_ACT10–ETL1_ACT12 in Fig. 4 or the activity ETL4_ACT5 in Fig. 13a).

Out transitive degree metrics capture the dependencies of a module with its various non-adjacent sources. These metrics exhibit more valuable results for activities, which act as "hubs" of various different paths from sources in complex workflows. For cases where the out-degree metrics *do not* provide a clear view of the evolution potential of two or more modules, the out-transitive degree and entropy metrics may offer a more adequate prediction (as for example ETL4_Q3 and ETL4_Q2 in Fig. 7a, d).

Hence, the module-level design of an ETL workflow is another crucial factor for the overall impact of evolution on the whole workflow. Thus, in terms of design, since attri-

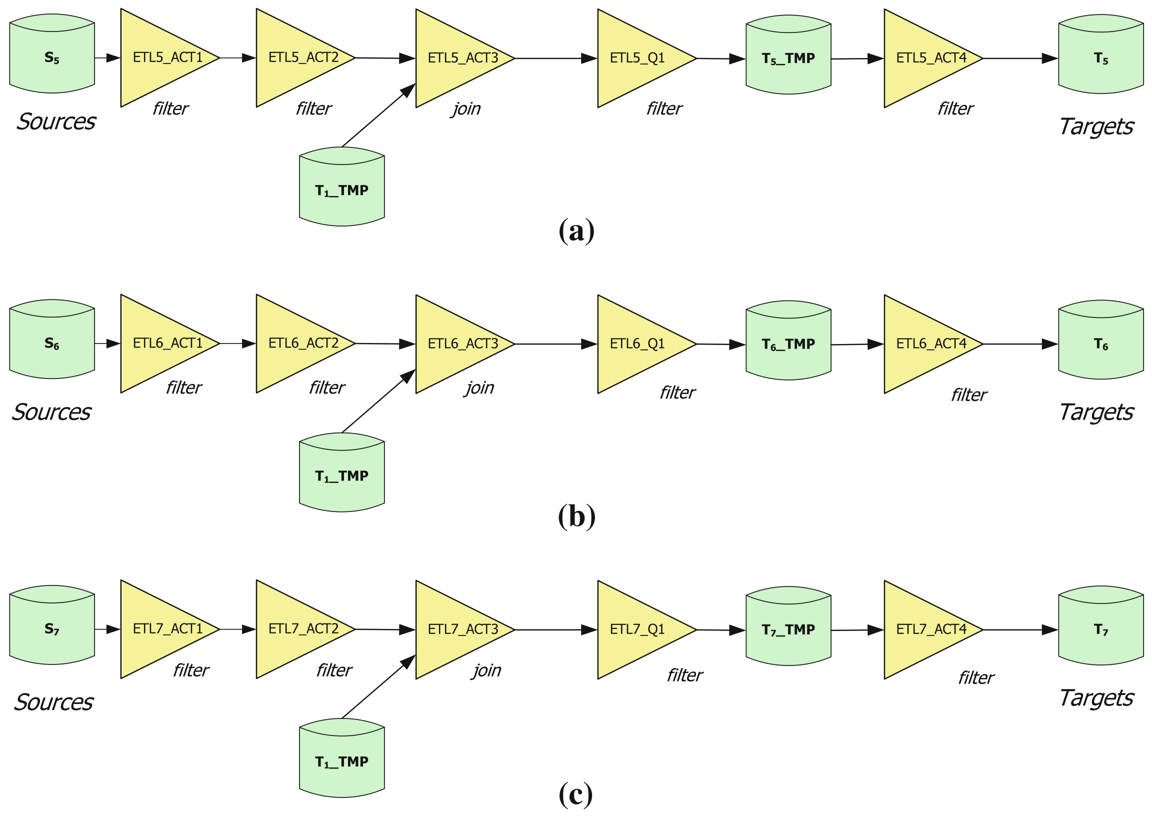


Fig. 14 **a** ETL5, **b** ETL6, **c** ETL7 workflows

bute reduction activities (e.g., projections, group by queries) are less likely to be affected by evolution actions than other activities that retain or increase the number of attributes in the workflow (many projections with joins), the ETL designer should attempt placing the attribute reduction activities in the early stages of the workflow in order to restrain the flooding of evolution events. In a way, that is in accordance with typical performance optimization strategies, where the most selective operations should be pushed toward the start of the flow.

Heuristics Based on these observations, we identify the most suitable metrics for each type of construct and provide possible optimization heuristics for reducing the maintenance effort. Table 6 reflects our observations. When persistent data stores are involved, the generic guideline is to retain their schema as small as possible. Since the schema size affects a lot the propagation of evolution events, it is advisable to reduce schema sizes across the ETL flow, so activities that help in that direction should be considered first. In addition, based on our discoveries related to what metric is suitable for each construct, e.g., transitive degree metrics are good predictors for modules with many providers and the out-degree and out strength metrics could be used for modules with a single provider.

Discussion Finally, we discuss how our methods and results may be used elsewhere and how such design choices may affect other ETL optimization objectives.

Generalization of results. Our analysis is based on a specific case study and the extent of how much this is representative is hard to show. However, in a previous work, we had presented a benchmark for ETL designs, where we presented a set of frequently used ETL template designs like butterfly, tree, fork, primary flow, and so on [35,36]. Interestingly, the designs in our case study resemble either those template designs or a combination of those. In particular, ETL1 is a complex *butterfly*-like design, ETL2 and ETL3 are *tree* designs, ETL4 is a combination of *fork* and *tree* designs, and ETL5, ETL6, and ETL7 are *primary flow* designs. Hence, we believe that the results obtained in this study may serve as general hints in other ETL projects as well.

Optimization trade-offs. In general, the aforementioned guidelines that favor maintainability of ETL flows contradict the normal practice for improving ETL performance. For example, when we have source data stores with large schema sizes, from an evolution handling perspective, it makes sense to split the schema into smaller chunks. How to efficiently do this for not hurting performance much (e.g., for avoiding join operations later on) is an open and challenging research

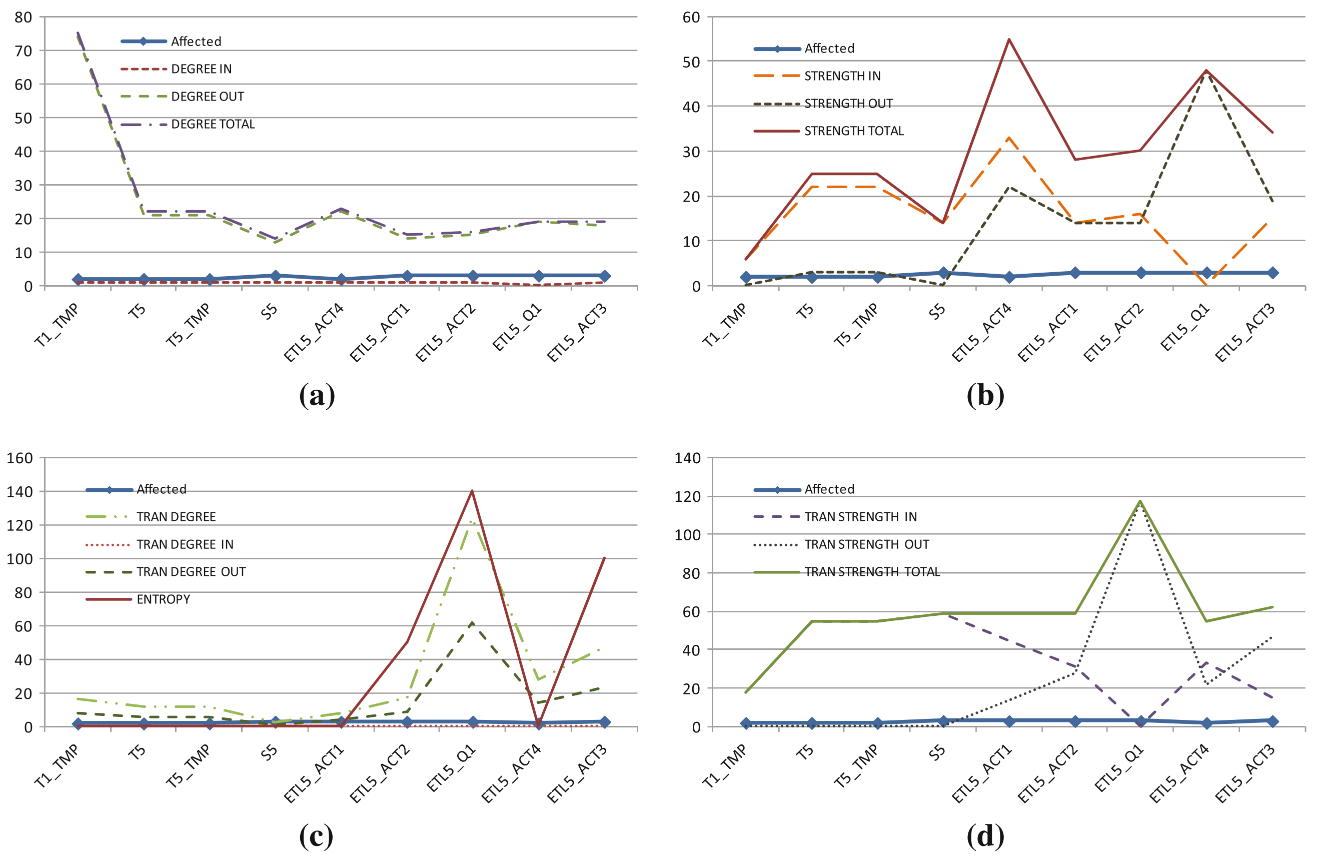


Fig. 15 Results for examined metrics for ETL5

question. As another example trade-off, although an attribute reduction operation like aggregation seems to better be placed at the beginning of a flow in terms of maintainability, in general, it should not be placed early in the flow when performance is considered. For performance, it makes sense to have early in the flow tuple reduction operators and heavier operations like aggregations should be placed next. Clearly, the ETL architect has to deal with an interesting trade-off between maintainability and performance, two very important quality factors for the design of an ETL system. However, this topic is out of the scope of this paper. Preliminary efforts towards such a multi-objective optimization of ETL flows has been presented elsewhere (e.g., [35–37]).

Usage in ETL engines Ideally, database administrators and ETL designers can employ these metrics for detecting, evaluating, and most importantly, experimenting with the design properties of ETL flows with respect to evolution. Based on such an analysis, the designer may decide to modify an ETL design or choose among more than one design for improving the maintainability of her system. In addition, the metric suite that we propose may be incorporated to an existing ETL tool for facilitating the ETL design. Since the most popular ETL tools already represent an ETL flow as a graph, measuring and predicting the evolution impact with metrics as those proposed in this work, is a realistic goal. Alternatively, the metric suite may be used as a basis of an external module—like our home-grown tool, Hecataeus—that could connect to an ETL tool. Then, based on such measures, the ETL designer could be notified about possible actions.

6 Related Work

Various approaches exist in the area of database metrics. Most of them attempt to define a set of database metrics and map them to abstract quality factors, such as maintainability, good database design, and so on. According to the model in which they are applied, we can categorize these efforts into conceptual metrics referring to the conceptual design of the database (i.e. ER diagram), relational metrics referring to the logical design of the database (i.e. relational data diagram), multidimensional metrics evaluating the design of data warehouses, information-theoretic approaches, etc.

Conceptual metrics are useful for evaluating quality issues for a database in the early stage of the design. To summarize the motivation for conceptual-level metrics, a "good" design

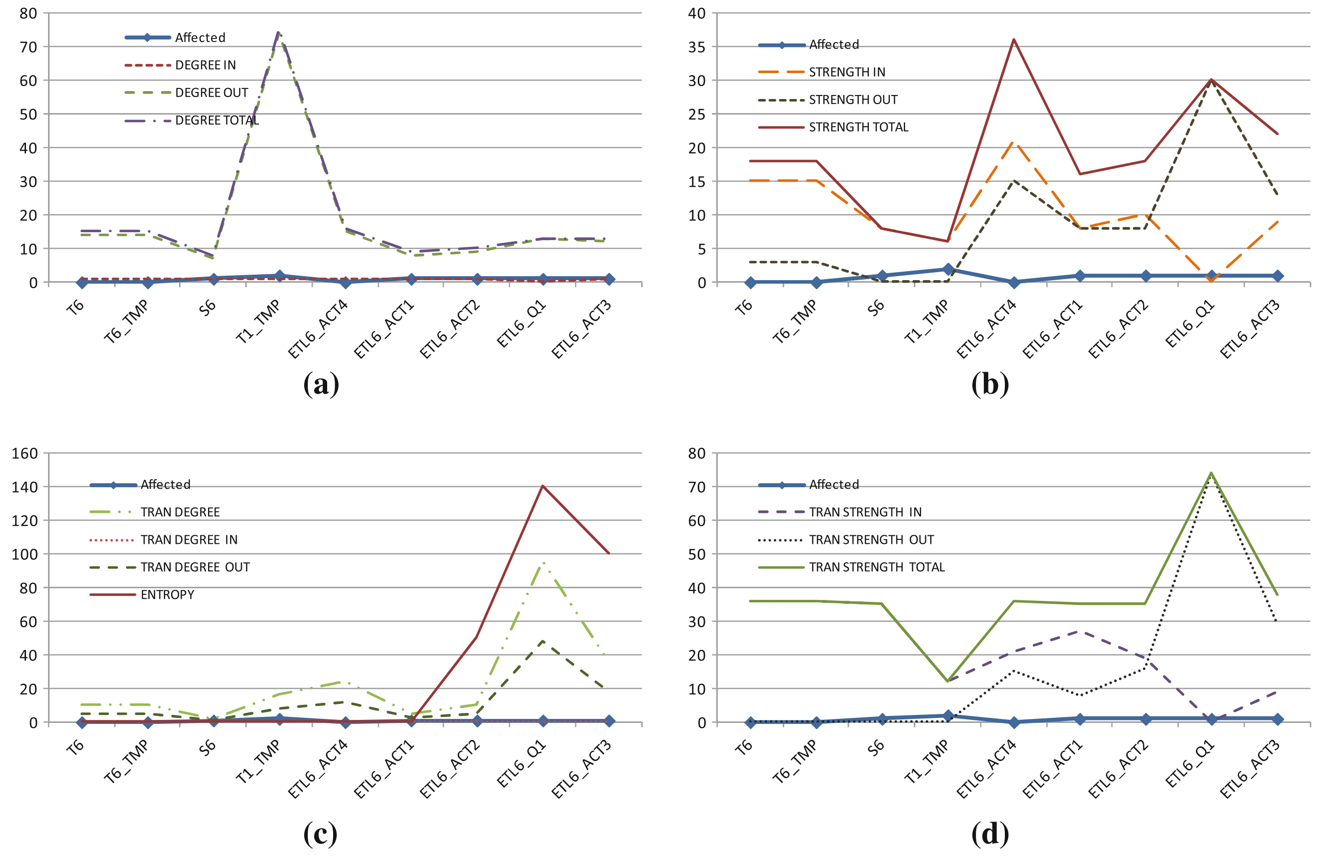


Fig. 16 Results for examined metrics for ETL6

Table 6 Metrics and heuristics for different types of ETL construct

ETL construct	Most suitable metric	Heuristic
Source tables	$D^{O}(v)$	Retain small schema size
Intermediate & target tables	$D^{O}(v)$	Retain small schema size in intermediate tables
Filtering activities	$D^{O}(v)$, $D^{Os}(v)$	Retain small number of conditions
Join activities	$D^{O}(v)$, $D^{Os}(v)$, $TD^{O}(v)$, $TD^{Os}(v)$, $H^{s}(v)$	Move to early stages of the workflow
Project activities	$D^{O}(v)$, $D^{Os}(v)$, $TD^{O}(v)$, $TD^{Os}(v)$, $H^{s}(v)$	Move attribute reduction activities to early stages of the workflow and attribute increase activities to later stages

at the conceptual level of a database may assure that fewer inconsistencies will emerge (mainly in terms of fundamental violations, e.g., primary and foreign keys) and furthermore fewer changes are needed during the lifetime of the information system, in general. In one of the early works, Gray et al. [16] propose two objective and open-ended metrics, namely ER metric and Area metric, to evaluate the quality of an ER diagram. ER Metric is a measure of the complexity of an ERD, based on the number of relationships between entities and Area metric is a measure of the compliance of an ERD with the corresponding ERD in 3rd Normal Form. Kesh [21] develops a method for assessing the quality of an ERD, based on both ontological and behavioral components. Ontological components are distinguished into structure and content metrics. Structure metrics are suitability, soundness, consistency, and conciseness, whereas content metrics are completeness, cohesiveness, and validity. Behavioral components are considered to be usability (from the user's point of view), usability (from the designer's point of view), maintainability, accuracy, and performance. Moreover, Moody [25] proposes a data model quality evaluation framework, which can be applied to a wide range of organizations. The proposed framework comprises a set of eight quality factors (completeness, integrity, flexibility, understandability, correctness, simplicity, integration, and implementability) which can be considered as properties of a data model with positive and negative interactions with each other. They are, in turn, evaluated by a set of 25 *quality metrics*. The quality

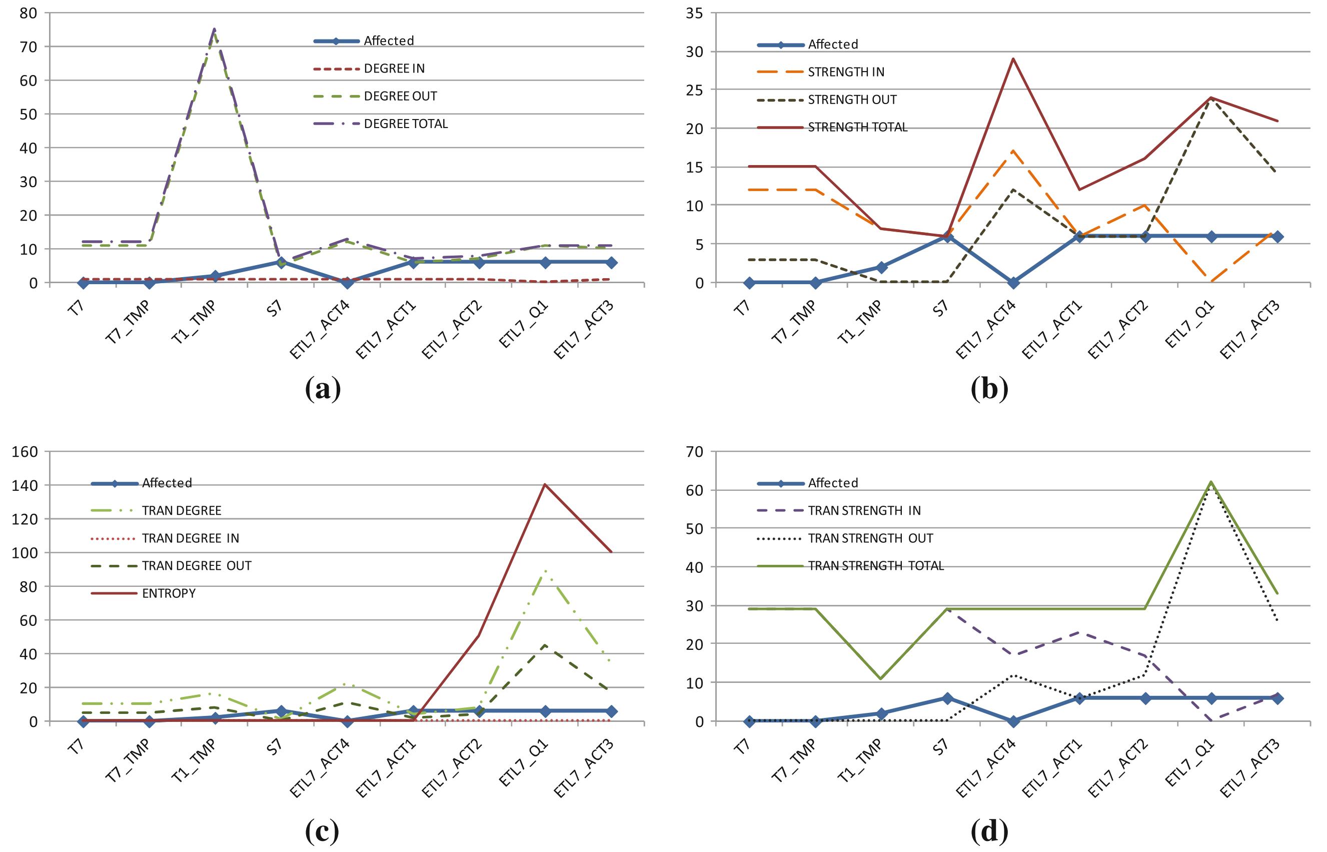


Fig. 17 Results for examined metrics for ETL7

factors may contribute to the overall quality of the system according to *weights*, which determine the importance of each factor in a problem situation. Genero et al. [13] focus on measuring the maintainability of ER diagrams through evaluating their structural complexity. They introduce a set of open-ended metrics and classify them into three main categories: entity metrics (i.e., number of entities within an ERD), attribute metrics (i.e., number of attributes within an ERD, number of composite attributes, etc.), and relationship metrics (i.e. number of M:N relationships, etc.). Wedemeijer [41] proposes a metric set for evaluating the stability capabilities of conceptual data model. The author sets up a framework for stability of conceptual schemas and proceeds to develop a set of metrics from it. The metrics are based on measurements of conceptual features, such as the number of conceptual constructs affected by a change, the complexity of a conceptual schema, the abstraction of a conceptual schema, etc. Last, Berenguer et al. [4] present a set of quality indicators and metrics for conceptual models of data warehouses. They employ UML diagrams for modeling multidimensional databases and in this context they define metrics for capturing diagram's properties such as number of packages in a diagram, number of relationships between two packages, etc. Although they provide a methodology for theoretically validating the proposed metric set, they do not present an empirical validation.

Relational database metrics are used as measures for the quality of a database at the logical level. Relational metrics are used to measure internal characteristics and structures of a database, such as tables, foreign keys, and so on. Normalization theory can give the guidelines for designing a database, but still cannot address other quality issues, such as the maintainability—or evolution—of a database. In [7,8,33], the authors propose a set of metrics for relational databases that focuses on assessing maintainability issues in a database, such as analyzability, testability, stability, and changeability. These are the number of relational tables (NT) in the database, the number of foreign keys (NFK), total number of attributes (NA), and the depth of referential tree (DRT), which is the maximum distance from a table towards another table through referential integrity constraints). Analyzability is proportionally correlated to NT, NA, DRT and NFK, changeability to NT, testability to NT, NA, and NFK, whereas stability is correlated to NT in an inverse relationship. Last, in [27], the authors propose a set of quality metrics, defined at four granularity levels (database, relation, attribute, and value) that measure referential completeness and consistency.

Data warehouse metrics. Quality in the context of data warehouses (DW) has been studied in [13,38]. The authors propose mathematical techniques for measuring or optimizing certain aspects of DW quality and adapt the Goal-Question-Metric approach from software quality management to a metadata management environment to link these special techniques to a generic conceptual framework of DW quality. Similar to aforementioned approaches DW quality is classified into several quality dimensions according to the stakeholders (e.g., data warehouse administrator, the programmer, and the decision maker) that are typically interested in them; each of these dimensions is further mapped to sample types of measurement (metrics), which help to establish the quality of a particular DW component with respect to a particular quality dimension. Various types of measurements are introduced to evaluate these dimensions. For example, the administrator is interested in the completeness dimension which concerns the preservation of all the crucial knowledge in the data warehouse schema (model), which is further quantified by the number of missing entities in the DW schema with respect to the conceptual model. Another approach to DW quality metrics is presented in [7,8]. They elaborate three kinds of metrics: table metrics regarding only table characteristics of the database (e.g. number of attributes, number of foreign keys), star metrics regarding only multidimensional characteristics of the database (e.g. number of dimension tables), and last, schema metrics regarding characteristics of the whole database schema (e.g. number of fact tables, number of overall dimension tables).

Data Warehouses and their Evolution. Concerning the evolution of data warehouses, we refer the reader to an excellent survey of [42]. A distinct line of work concerns multi-version data warehousing: see for example, Wrembel and Morzy [43] that handle the evolution of data warehouses via multiple versions and [2] that handles the problem via nested transactions, or Golfarelli et al. [14] for cross-version querying. Another excellent survey by [15] on temporal data warehousing contains a summary of the related work along these lines. A survey by [40] discusses the area of ETL and the related work.

Various *information-theoretic metrics* exist in software engineering for evaluating the quality of software design [1,18,22]. In the data management field, an information theoretic approach to evaluating the design quality of data warehouses is presented in [23], where the relation between entropy and redundancy in the context of data warehouses is studied. They show that the redundancy in the snowflake join of the primary key of the fact table is zero, i.e. it is minimal. They define a new normal form, namely SSNF—Snowflake Schema Normal Form, justifying it in terms of entropy-based equations.

Most of the aforementioned approaches consider design metrics that correlate structural properties of the database schema to abstract quality factors. However, they confine themselves to constructs internal to the database without taking into account the incorporation of constructs surrounding the database. To the best of our knowledge this is the first set of design metrics that are explicitly targeted towards the assessment of evolution ability of the design of a data-centric ecosystem as a whole to evolutionary processes.

Formally specified frameworks. Several *software quality metrics* have been introduced in the software engineering community. Software measurement is a well-established research area that has been explored under many different programming paradigms (e.g., procedural, object oriented, service oriented, etc.) and for various stages of the lifecycle of software development (i.e., requirements analysis, design, coding, testing and maintenance). A detailed presentation of software metrics, software quality factors, and measurement approaches is out of the scope of this paper and can be found in [12]; still, we mention here the concepts of module cohesion and coupling that are mostly used for assessing the maintainability of software [24] along with complexity, as well (referred as the 3 'c'-s in [34]).

Briand et al. [6] employed measurement theory to provide a set of five generic categories of measures for software artifacts. In a previous work, we made a first attempt to relate these families of measures to ETL flows [39]. In that work, we used a different model for module representation (based on LDL) and formally proved that the measures proposed respect the properties of the framework by [6]. However, representing ETL activities with LDL does not scale well in terms of operations that can be supported. In addition, a typical, modern ETL flow involves operations implemented in different environments and runs on different engines (e.g., operations in Java, PL/SQL, SQL, Perl, Awk, etc. that may run in different engines like DBMS, ETL, Map-Reduce, and so on). However, independently of the internal representation, our graph-based model for data-centric systems, such as ETL flows, can be viewed as a modular system, with queries, views and relations being its building blocks encapsulating data and business logic. We generalize thus the discussion, to highlight how our method fits within a more formally specified framework like the one by [6]. First, we start by referring to the involved measures, which are

- *Size*, referring to the number of entities that constitute the software artifact; we assess the size of a (sub)graph by the number of its nodes.
- *Length*, referring to the longest path of relationships among these entities, which we assessed by the maximum transitive dependency of a module's node.
- *Complexity*, referring to the amount of inter-relationships of a component, which we assessed measured by the number of internal edges plus the 50 % of the strength of the module.

- *Coupling*, capturing the amount of interrelationships between the different modules of a system, which we assessed by the strength of a module.
- *Cohesion*, measuring the extent to which each module performs exactly one job, by evaluating how closely related are its components, which we assessed as the fraction of the input/output nodes of a module related to some internal function of the module.

Some (but not all) of these metrics are straightforwardly related to the metrics used in this study. The transitive degree is an attempt to test the sensitivity of nodes depending on the length of their provider path. The coupling of modules within the flow is probably the most straightforward measure relating to the strength of the module.

Cohesion refers to how much interrelated are the constituents of a module. If a module performs more than one "job" and its constituents are divided in two groups of nodes doing different things, then a module is not cohesive. Still, related to data-centric software, cohesion is a weakly definable notion. For example, how can one *intuitively* convince on a model for the cohesion of a relation? In a more subtle line, even if we adopt the idea that each selection and each group-by constitutes a different "job" how convincing is it to assume that a query combining several selections and/or a group by is not cohesive?

A serious observation (that goes well beyond the scope of a case study) is that the Briand et al. meta-measures were originally thought towards traditional imperative/object-oriented software with loops and control structures rather than data-centric software. In [39], the authors provide measures for the Briand et al. [6] classification, but they do not fit important measures into the framework, like the maximum path over the graph or the degree of an individual attribute of a relation. Complexity is a good example where the respective notion in traditional software (McCabe's cyclomatic complexity) is not adequately mapped to a measure for data intensive software.

Hence, overall, we find that our most valuable metric, out-degree, which is going down to the details of individual attributes, does not fit well with the Briand et al. framework. At the same time, although module coupling is smoothly covered by strength, cohesion and complexity must be re-evaluated when we think of data-centric software.

7 Conclusions

In this paper, we have presented a real-world case study of data warehouse evolution for exploring the behavior of a set of metrics that (a) monitor the vulnerability of warehouse modules to future changes and (b) assess the quality of various ETL designs with respect to their maintainability. We have described first our graph-theoretic model for capturing the evolution impact in the ETL ecosystem, and then, we presented a detailed description of our metric suite. Finally, we have reported on our exhaustive, 6-month experimentation with real-world evolution scenarios affecting seven ETL workflows.

We have identified the schema size and module complexity as two important factors for the vulnerability of a system. We have observed that out-degrees help as predictors for the source tables; the out-degree and out-strength are very good predictors for the evolution of views; out transitive degree and entropy may be applied for queries in addition to the aforementioned metrics. Based on our experiments, we have compiled a list of lessons learned regarding the evolution behavior of an ETL environment with respect to the schema of the source tables, its constituent activities, and its overall design. We believe that these metrics and the lessons learned in this paper can be practically useful for database administrators and designers for detecting vulnerable parts and evaluating the design properties of data-centric ecosystems, like ETL workflows, with respect to evolution.

Coming back to our starting point, have we answered the fundamental questions like *How good is an ETL design?* and *What makes an ETL design good or bad?*. We have demonstrated only ways to predict vulnerability to change and discussed some interrelationship with other aspects, like for example, performance. A complete answer to the above questions and an attempt to combine different aspects of the environments design (performance, vulnerability to change, understandability, etc.) in a comprehensive framework are prominent directions for future research. Another direction for future work concerns models that are not founded on a graph-based model. Although our approach is founded on a simple and intuitive graph representation of modules and their dependencies, it is quite possible that other approaches that avoid the translation of code to graphs can be pursued. How this can be done and what is the effect to the metrics used are open problems.

Acknowledgments We would like to thank the anonymous reviewers of a first version of this paper for their constructive comments that improved the clarity and completeness of the paper.

References

1. Allen EB (2002) Measuring graph abstractions of software: an information-theory approach. In: Proceedings of the 8th international symposium on software metrics (METRICS'02)
2. Bebel B, Królikowski, Z, Wrembel R (2006) Managing evolution of data warehouses by means of nested transactions (ADVIS'06)
3. Bellahsene Z (2002) Schema evolution in data warehouses. Knowl Inf Syst 4(2):283–304

4. Berenguer G, et al (2005) A set of quality indica-tors and their corresponding metrics for conceptual models of data warehouses. In: 7th International conference on data warehousing and knowledge discovery (DaWaK'05)
5. Blaschka M, Sapia C, Höfling G (1999) On schema evolution in multidimensional databases. In: 1st International conference on data warehousing and knowledge discovery (DaWaK'99)
6. Briand LC, Morasca S, Basili VR (1996) Property-based software engineering measurement. IEEE Trans Softw Eng 22(1):68–85
7. Calero C, Piattini M, Genero M (2001) Empirical validation of referential integrity metrics. Inf Softw Technol 43(15):949–957
8. Calero C, Piattini M, Pascual C, Serrano M (2001) Towards data warehouse quality metrics. In: Proceedings of the 3rd international workshop on design and management of data warehouses (DMDW'01)
9. Cleve A, Brogneaux A, Hainaut J (2010) A conceptual approach to database applications evolution. In: Proceedings of the 29th international conference on conceptual modeling (ER'10)
10. Fan H, Poulovassilis A (2004) Schema evolution in data warehousing environments—a schema transformation-based approach. In: Proceedings of the 23rd international conference on conceptual modeling (ER'04)
11. Favre C, Bentayeb F, Boussaid O (2007) Evolution of data warehouses' optimization: a workload perspective. In: 9th International conference on data warehousing and knowledge discovery (DaWaK'07)
12. Fenton NE, Pfleeger SL (1998) Software metrics: a rigorous and practical approach, revised 2nd edn. PWS Publishing Co.
13. Genero M, Piattini M, Calero C, Serrano M (2000) Measures to get better quality databases. In: Proceedings of the 2nd international conference on enterprise information systems (ICEIS'00)
14. Golfarelli M, Lechtenbörger J, Rizzi S, Vossen G (2006) Schema versioning in data warehouses: enabling cross-version querying via schema augmentation. Data Knowl Eng 59(2):435–459
15. Golfarelli M, Rizzi S (2009) A survey on temporal data warehousing. In: Database technologies: concepts, methodologies, tools, and applications, pp 221–237
16. Gray R, Carey B, McGlynn N, Pengelly A (1991) Design metrics for database systems. BT Technol J 9(4):69–79
17. Gupta A, Mumick IS, Rao J, Ross KA (2001) Adapting materialized views after redefinitions: techniques and a performance study. Inf Syst 26(5):323–362
18. Harrison W (1992) An entropy-based measure of software complexity. IEEE Trans Softw Eng 18(11):1025–1034
19. Inmon WH (2000) The data warehouse budget. White paper
20. Jarke M, Jeusfeld MA, Quix C, Vassiliadis P (1999) Architecture and quality in data warehouses: an extended repository approach. Inf Syst 24(3):229–253
21. Kesh S (1995) Evaluating the quality of entity relationship models. Inf Softw Technol 37(12):681–689
22. Kim K, Shin Y, Wu C (1995) Complexity measures for object-oriented program based on the entropy. In: Proceedings of the 2nd Asia-Pacific software engineering conference (APSEC '95)
23. Levene M, Loizou G (2003) Why is the snowflake schema a good data warehouse design?. Inf Syst 28(3):225–240
24. Lorenz M, Kidd J (1994) Object-oriented software metrics. Prentice Hall, Englewood Cliffs
25. Moody DL (1998) Metrics for evaluating the quality of entity relationship models. In: Proceedings of the 17th international conference on conceptual modeling (ER'98)
26. Nica A, Lee AJ, Rundensteiner EA (1998) The CSV algorithm for view synchronization in evolvable large-scale information systems. In: Proceedings of the 6th international conference on extending database technology (EDBT'98)
27. Ordonez C, García-García J (2008) Referential integrity quality metrics. Decis Support Syst 44(2):495–508
28. Papastefanatos G, Vassiliadis P, Simitsis A, Vassiliou Y (2008) Design metrics for data warehouse evolution. In: Proceedings of the 27th international conference on conceptual modeling (ER'08)
29. Papastefanatos G, et al (2008) Language extensions for the automation of database schema evolution. In: Proceedings of the 14th international conference on enterprise information systems (ICEIS'08)
30. Papastefanatos G, Vassiliadis P, Simitsis A, Vassiliou Y (2009) Policy-regulated management of ETL evolution. J Data Semantics 13:147–177
31. Papastefanatos G, Vassiliadis P, Simitsis A, Vassiliou Y (2010) HECATAEUS. Regulating schema evolution. In: Proceedings of the 26th IEEE international conference on data engineering (ICDE'10)
32. Papoulis A (1990) Probability & statistics. Prentice Hall, Englewood Cliffs
33. Piattini M, Genero M, Calero C (2001) Table oriented metrics for relational databases. Softw Quality J 9(2):79–97
34. Pressman RS, Ince D (2000) Software engineering (a practitioner's approach), 5th edn. European Adaptation. McGraw Hill
35. Simitsis A, Vassiliadis P, Dayal U, Karagiannis A, Tziovara V (2009) Benchmarking ETL workflows. In: Proceedings of the TPC technology conference (TPCTC'09)
36. Simitsis A, Wilkinson K, Castellanos M, Dayal U (2009) QoX-driven ETL design: reducing the cost of ETL consulting engagements. In: Proceedings of the 35th SIGMOD international conference on management of data (SIGMOD'09)
37. Simitsis A, Wilkinson K, Dayal U, Castellanos M (2010) Optimizing ETL workflows for fault-tolerance. In: Proceedings of the 26th IEEE international conference on data engineering (ICDE'10)
38. Vassiliadis P, Bouzeghoub M, Quix C (2000) Towards quality-oriented data warehouse usage and evolution. Inf Syst 25(2):89–115
39. Vassiliadis P, Simitsis A, Terrovitis M, Skiadopoulos S (2005) Blueprints and measures for ETL workflows. In: Proceedings of 24th international conference on conceptual modeling (ER 2005), 24–28 Oct 2005, Klagenfurt, Austria
40. Vassiliadis P (2009) A survey of extract–transform–load technology. Int J Data Warehousing Mining 5(3):1–27
41. Wedemeijer L (2000) Defining metrics for conceptual schema evolution. In: Proceedings of the 9th international workshop on foundations of models and languages for data and objects (FMLDO'00)
42. Wrembel R (2009) A survey of managing the evolution of data warehouses. Int J Data Warehousing Mining 5(2):24–56
43. Wrembel R, Morzy T (2006) Managing and querying versions of multiversion data warehouse (EDBT'06)

Open-Source Databases: Within, Outside, or Beyond Lehman's Laws of Software Evolution?

Ioannis Skoulis, Panos Vassiliadis, and Apostolos Zarras

Dept. of Computer Science and Engineering
University of Ioannina (Hellas)
{iskoulis, pvassil, zarras}@cs.uoi.gr

Abstract. Lehman's laws of software evolution is a well-established set of observations (matured during the last forty years) on how the typical software systems evolve. However, the applicability of these laws on databases has not been studied so far. To this end, we have performed a thorough, large-scale study on the evolution of databases that are part of larger open source projects, publicly available through open source repositories, and report on the validity of the laws on the grounds of properties like size, growth, and amount of change per version.

Keywords: Schema evolution, software evolution, Lehman's laws

1 Introduction

Software evolution is the change of a software system over time, typically performed via a remarkably difficult, complicated and time consuming process, software maintenance. In an attempt to understand the mechanics behind the evolution of software and facilitate a smoother, lest disruptive maintenance process, Meir Lehman and his colleagues introduced a set of rules in mid seventies [1], also known as the *Laws on Software Evolution* (Sec. 2). Their findings, that were reviewed and enhanced for nearly 40 years [2], [3], have, since then, given an insight to managers, software developers and researchers, as to *what* evolves in the lifetime of a software system, and *why* it does so. Other studies ([4], [5], [6] to name a few significant ones) have complemented these insights in this field, typically with particular focus to open-source software projects.

In sharp distinction to traditional software systems, database evolution has been hardly studied throughout the entire lifetime of the data management discipline. This deficit in our knowledge is disproportional to the severity of the implications of database evolution, and in particular, of database schema evolution. A change in the schema of a database may immediately drive surrounding applications to crash (in case of deletions or renamings) or be semantically defective or inaccurate (in the case of information addition, or restructuring). Overall, schema evolution threatens the syntactic and semantic validity of the surrounding applications and severely affects both developers and end-users. Given this importance, it is only amazing to find out that in the past 40 years of database research, only three(!) studies [7], [8] and [9] have attempted a first step towards understanding the mechanics of schema evolution. Those studies, however, focus on the statistical properties of the evolution and do not provide details on the actual events, or the mechanism that governs the evolution of database schemata.

In this paper, *we perform the first large-scale study of schema evolution in the related literature. Specifically, we study the evolution of the logical schema of eight databases, that are parts of publicly available, open-source software projects.* To achieve the above goal, we have collected, cleansed and processed the available versions of the database schemata for the eight case studies. Moreover, we have extracted the changes that have been performed in these versions and, finally, we have come up with the respective datasets that can serve as a foundation for future analysis by the research community (Sec. 3). Concerning the applicability of Lehman's laws to open-source databases, our results show that *the essence of Lehman's laws holds*: evolution is not about uncontrolled growth; on the contrary, there appears to be a stabilization mechanism that employs perfective maintenance to control the otherwise growing trend of increase in information capacity of the database (Sec. 4). Having said that, we also observe that the growth mechanisms and the change patterns are quite different between open source databases and typical software systems.

2 Lehman Laws of Software Evolution in a Nutshell

Meir M. Lehman and his colleagues, have introduced, and subsequently amended, enriched and corrected a set of rules on the behavior of software as it evolves over time [1], [2], [3]. Lehman's laws focus on *E-type systems*, that concern "software solving a problem or addressing an application in the real-world" [2]. The main idea behind the laws of evolution for E-type software systems is that their *evolution is a process that follows the behavior of a feedback-based system.* Being a feedback-based system, the evolution process has to balance (a) *positive feedback*, or else the need to adapt to a changing environment and grow to address the need for more functionality, and, (b) *negative feedback*, or else the need to control, constrain and direct change in ways that prevent the deterioration of the maintainability and manageability of the software. In the sequel we list the definitions of the laws as they are presented in [3], in a more abstract form than previous versions and with the benefit of retrospect, after thirty years of maturity and research findings.

(I) Law of Continuing Change An E-type system must be continually adapted or else it becomes progressively less satisfactory in use.

(II) Law of Increasing Complexity As an E-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.

(III) Law of Self Regulation Global E-type system evolution is feedback regulated.

(IV) Law of Conservation of Organisational Stability The work rate of an organisation evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime.

(V) Law of Conservation of Familiarity In general, the incremental growth (growth ratio trend) of E-type systems is constrained by the need to maintain familiarity.

(VI) Law of Continuing Growth The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.

(VII) Law of Declining Quality Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining.

(VIII) Law of Feedback System E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

Before proceeding with our study, we present a first apodosis of the laws, taking into consideration both the wording of the laws, but most importantly their accompanying explanations [3].

An E-Type software system continuously changes over time (I) obeying a complex feedback-based evolution process (VIII). On the one hand, due to the need for growth and adaptation that acts as positive feedback, this process results in an increasing functional capacity of the system (VI), produced by a growth ratio that is slowly declining in the long term (V). The process is typically guided by a pattern of growth that demonstrates its self-regulating nature: growth advances smoothly; still, whenever there are excessive deviations from the typical, baseline rate of growth (either in a single release, or accumulated over time), the evolution process obeys the need for calibrating releases of perfective maintenance (expressed via minor growth and demonstrating negative feedback) to stop the unordered growth of the system's complexity (III). On the other hand, to regulate the ever-increasing growth, there is negative feedback in the system controlling both the overall quality of the system (VII), with particular emphasis to its internal quality (II). The effort consumed for the above process is typically constant over phases, with the phases disrupted with bursts of effort from time to time (IV).

3 Experimental Setup of the Study

Datasets. We have studied eight database schemata from open-source software projects. *ATLAS*[1] is a particle physics experiment at CERN, with the goal of learning about the basic forces that have shaped our universe – famously known for the attempt on the Higgs boson. *BioSQL*[2] is a generic relational model covering sequences, features, sequence and feature annotation, a reference taxonomy, and ontologies from various sources such as GenBank or Swissport. *Ensembl* is

[1] http://atlas.web.cern.ch/Atlas/Collaboration/
[2] http://www.biosql.org/wiki/Main_Page

a joint scientific project between the European Bioinformatics Institute (EBI)[3] and the Wellcome Trust Sanger Institute (WTSI)[4] which was launched in 1999 in response to the imminent completion of the Human Genome Project. The goal of Ensembl was to automatically annotate the three billion base pairs of sequences of the genome, integrate this annotation with other available biological data and make all this publicly available via the web. *MediaWiki*[5] was first introduced in early 2002 by the Wikimedia Foundation along with Wikipedia, and hosts Wikipedia's content since then. *Coppermine*[6] is a photo gallery web application. *OpenCart*[7] is an open source shopping cart system. *PhpBB*[8] is an Internet forum package. *TYPO3*[9] is a free and open source web content management framework.

Dataset Collection and Processing. A first collection of links to available datasets was made by the authors of [9], [10][10]; for this, these authors deserve honorable credit. We isolated eight databases that appeared to be alive and used (as already mentioned, some of them are actually quite prominent). For each dataset we gathered as many schema versions (DDL files) as we could from their public source code repositories (cvs, svn, git). We have targeted main development branches and trunks to maximize the validity of the gathered resources. *We are interested only on changes of the database part of the project as they are integrated in the trunk of the project.* Hence, we collected all the versions of the database, committed at the trunk or master branch, and ignored all other branches of the project.

We collected the files during June 2013. For all of the projects, we focused on their release for MySQL (except ATLAS Trigger, available only for Oracle). The files were then processed by sequential pairs from our tool, Hecate, that allows the detection of (a) changes at the attribute level, and specifically, attributes inserted, deleted, having a changed data type, or participation in a changed primary key, and (b) changes at the relation level, with relations inserted and deleted, in a fully automated way. Hecate, was then used to give us (a) the differences between two subsequent committed versions, and (b) the measures we needed to conduct this study – for example, the *size of the schema* (in number of tables and attributes), the total number of changes for each transition from a version to the next, which we also call *heartbeat*, or the *growth* assessed as the difference in the size of the schema between subsequent versions. *Hecate, along with all the data sets and our results are available at our group's public repository* https://github.com/DAINTINESS-Group.

[3] https://www.ebi.ac.uk/
[4] https://www.sanger.ac.uk/
[5] https://www.mediawiki.org/wiki/MediaWiki
[6] http://coppermine-gallery.net/
[7] http://www.opencart.com
[8] https://www.phpbb.com/
[9] http://typo3.org/
[10] http://data.schemaevolution.org

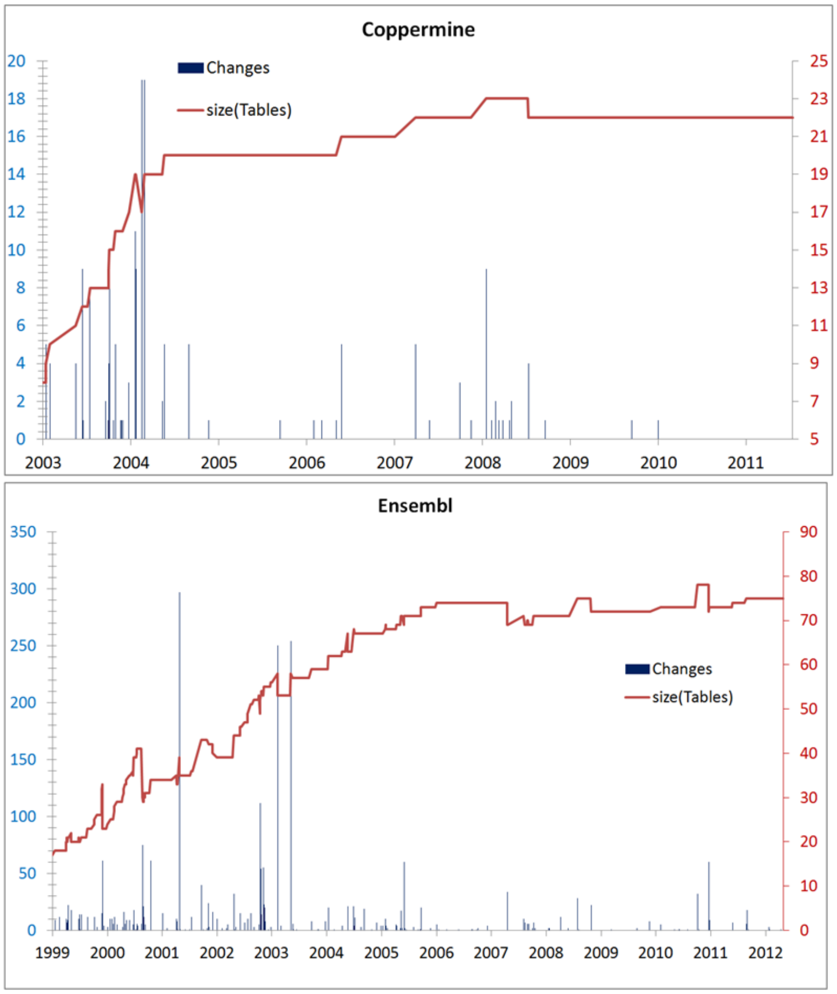


Fig. 1. Combined demonstration of heartbeat (number of changes per version) and schema size (no. of tables) for Coppermine and Ensemble. The left axis signifies the amount of change and the right axis the number of tables.

4 Assessing the Laws for Schema Evolution

The laws of software evolution where developed and reshaped over forty years. Explaining each law in isolation from the others is precarious as it risks losing the deeper essence and inter-dependencies of the laws [3]. To this end, in this section, we organize the laws in three thematic areas of the overall evolution management mechanism that they reveal. The first group of laws discusses the existence of a feedback mechanism that constrains the uncontrolled evolution of software. The second group discusses the properties of the growth of the system,

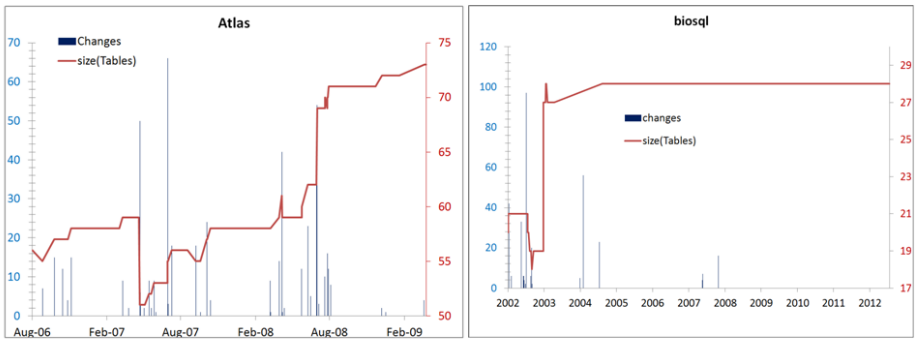


Fig. 2. Combined demonstration of heartbeat (number of changes per version) and schema size (no. of tables) for Atlas and BioSQL. The left axis signifies the amount of change and the right axis the number of tables.

i.e., the part of the evolution mechanism that accounts for positive feedback. The third group of laws discusses the properties of perfective maintenance that constrains the uncontrolled growth, i.e., the part of the evolution mechanism that accounts for negative feedback.

4.1 Is There a Feedback-based System for Schema Evolution?

Law of continuing change (Law I) The main argument of the first law is that the schema continuously changes over time. To validate the hypothesis that the law of continuing change holds, we study the heartbeat of the schema's life (see Fig. 1 and 2 for a combined demonstration of heartbeat and schema size).

With the exception of BioSQL that appeared to be "sleeping" for some years and was later re-activated, in all other cases, we have changes (sometimes moderate, sometimes even excessive) over the entire lifetime of the database schema. An important observation stemming from the visual inspection of our change-over-time data, is that the term *continually* in the law's definition is challenged: *we observe that database schema evolution happens in bursts, in grouped periods of evolutionary activity, and not as a continuous process*! Take into account that the versions with zero changes are versions where either commenting and beautification takes place, or the changes do not refer to the information capacity of the schema (relations attributes and constraints) but rather, they concern the physical level properties (indexes, storage engines, etc) that pertain to performance aspects of the database.

Can we state that this stillness makes the schema "unsatisfactory" (referring back to the wording of the first law by Lehman)? We believe that the answer to the question is negative: since the system hosting the database continues to be in use, user dissatisfaction would actually call for continuous growth of the database, or eventual rejection of the system. This does not happen. On the other hand, our explanation relies on the reference nature of the database in terms of software architecture: if the database evolves, the rest of the code, which is basically using the database (and not vice versa), breaks!

Overall, if we account for the exact wording of the law, we conclude that the law partially holds.

Law of feedback system (Law VIII) The wording of Law VIII refers to the existence of a self-stabilizing feedback mechanism that governs evolution. Its experimental evaluation typically refers to the possibility of demonstrating adherence to a basic formula of feedback, by estimating the size of the system (here: in terms of number of relations) accurately – i.e., with small error compared to the actual values. The formula typically used [2] is: $\widehat{S}_i = \widehat{S}_{i-1} + \frac{\overline{E}}{\widehat{S}^2_{i-1}}$, where $\widehat{S}$ refers to the estimated system size and $\overline{E}$ is a model parameter approximating effort (actually obtained as the average value of a set of past assessments of E).

Related literature [2] suggests computing $\overline{E}$ as the average value of individual E_i, one per transition. Then, we need to estimate these individual effort approximations. [2] suggests two formulae that we generalize here as follows: $E_i = \frac{s_i - s_\alpha}{\sum_{j=\alpha}^{i-1} \frac{1}{s_j^2}}$, where s_i refers to the actual size of the schema at version i and α refers to the version from which counting starts. Specifically, [2] suggests two values for α, specifically (i) 1 (the first version) and (ii) s_{i-1} (the previous version).

We now move on to discuss what seems to work and what not for the case of schema evolution. We will use the OpenCart data set as a reference example; however, all datasets demonstrate exactly the same behavior.

First, we assessed the formulae of [2]. In this case, we compute the average $\overline{E}$ of the individual E_i over the entire dataset. We employ four different values for α, specifically 1, 5, 10, and n, with n being the entire data set size, and depict the result in Fig. 3, where the actual size is represented by the blue solid line. The results indicate that the approximation modestly succeeds in predicting an overall increasing trend for all four cases, and, in fact, all four approximations targeted towards predicting an increasing tendency that the actual schema does not demonstrate. At the same time, all four approximations fail to capture the individual fluctuations within the schema lifetime.

A better estimation occurred when we realized that back in 1997 people considered that the parameter $\overline{E}$ was constant over the entire lifetime of the project; however, later observations (see [3]) led to the revelation that the project was split in phases. So, for every version i, we compute $\overline{E}$ as an average over the last τ E_j values, with small values for τ (1/5/10).

As we can see in Fig. 3, *the idea of computing the average $\overline{E}$ with a short memory of 5 or 10 versions produced extremely accurate results. This holds for all data sets.* This observation also suggests that, if the phases that [3] mentioned actually exist for the case of database schema, they are really small and a memory of 5-10 versions is enough to produce very accurate results.

Overall, *the evolution of the database schema appears to obey the behavior of a feedback-based mechanism, as the schema size of a certain version of the*

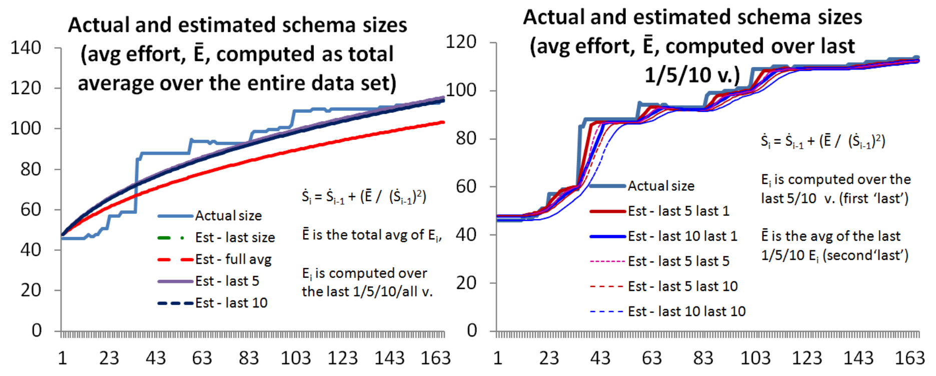


Fig. 3. Actual and estimated schema size via a total (left) and a bounded (right) average of individual E_i for OpenCart; the x-axis shows the version id.

database can be accurately estimated via a regressive formula that exploits the amount of changes in recent, previous versions.

Law of self-regulation (Law III). Whereas the law simply states that the evolution of software is feedback regulated, its experimental validation in the area of software systems is typically supported by the observation of a recurring pattern of smooth expansion of the system's size(a.k.a. "baseline" growth), that is interrupted with releases of perfective maintenance with size reductions or with releases of growth. Moreover, due to a previous wording of the law (e.g., see [2]) that described change to follow a normal distribution, the experimental assessment included the validation of whether growth demonstrates oscillations around the average value [1,2,3].

Size. The evolution of size can be observed in Fig. 1 and 2. We have to say that we simply do not detect the same behaviour that Lehman did (contrast Fig. 1, 2 to the respective figures of articles [1] and [2]): in sharp contrast to the smooth baseline growth that Lehman has highlighted, the evolution of the size of the studied database schemata provides a landscape with a large variety of *sequences of the following three fundamental behaviors.*

- In all schemata, we can see periods of increase, especially at the beginning of their lifetime or after a large drop in the schema size. This is an indication of positive feedback, i.e., the need to expand the schema to cover the information needs of the users.
- In all schemata, there are versions with drops in schema size. Those drops are typically sudden and steep and usually take place in short periods of time. Sometimes, in fact, these drops are of significantly larger size than the typical change. We can safely say that the existence of these drops in the schema size indicate perfective maintenance and thus, the existence of a negative feedback mechanism in the evolution process.

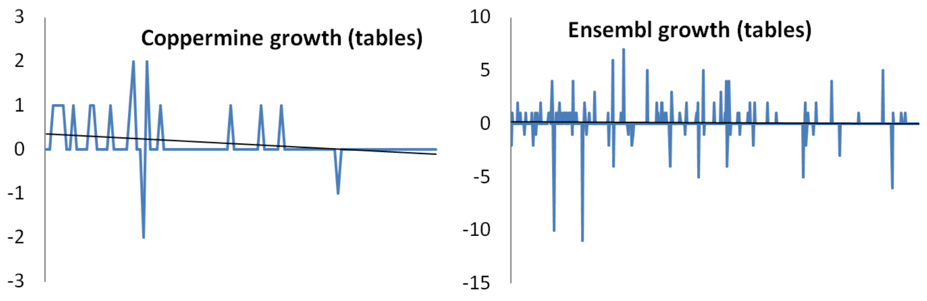


Fig. 4. Growth for Coppermine and Ensembl (over version id, concealed for fig. clarity); the slowly dropping solid line shows the linear interpolation of the values

– In all schemata, there are periods of stability (i.e., size stays still, or –near-still).

Growth. Growth (i.e., the difference in the size between two subsequent versions) in all datasets has the following broad characteristics (Fig. 4, 6). In terms of tables, in most cases, growth is small (typically ranging within 0 and 1), and moderately small when it comes to attributes. We *have too many occurrences of zero growth*, typically iterating between small non-zero growth and zero growth. Due to perfective maintenance, we also have negative values of growth (less than the positive ones). We do not have a constant flow of versions where the schema size is continuously changing; rather, we have small spikes between one and zero. Thus, we have to state that the growth comes with *a pattern of spikes*. Due to this characteristic, *the average value is typically very close to zero (on the positive side) in all datasets, both for tables and attributes*. There are *few cases of large change* too; we forward the reader to Law V for a discussion of their characteristics.

We would like to put special emphasis to the observation that *change is small*. In terms of tables, growth is mostly bounded in small values. This is not directly obvious in the charts, because they show the ripples; however, almost all numbers are in the range of [-2..2] – in fact, mostly in the range [0..2]. Few abrupt changes occur. In terms of attributes, the numbers are higher, of course, and depend on the dataset. Typically those values are bounded within [-20,20]. However, the deviations from this range are not many.

In the course of our deliberations, we have observed a pattern common in all datasets: *there is a Zipfian model in the distribution of frequencies*. Observe Fig. 5 that comes with two parts, both depicting how often a growth value appears in the attributes of Ensemble. The x-axis keeps the delta size and the y-axis the number of occurrences of this delta. In the left part we include zeros in the counting (343 occurrences out of 528 data points) and in the right part we exclude them (to show that the power law does not hold only for the most popular value). We observe that there is a small range of deltas, between -2 and

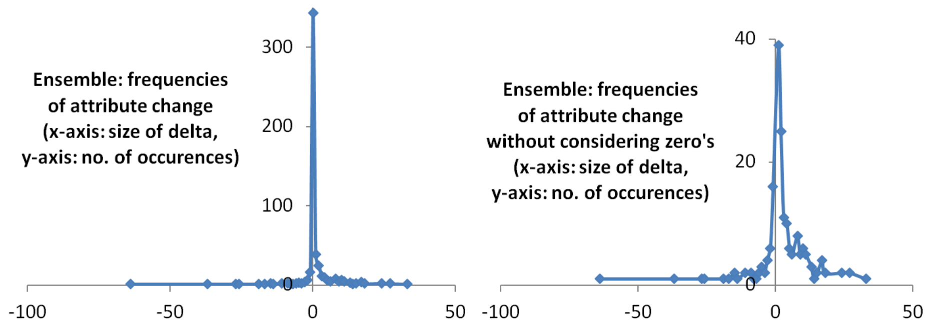


Fig. 5. Frequency of change values for Ensembl attributes

4 that takes up 450 changes out of the 528. This means that, despite the large outliers, change is strongly biased towards small values close to zero.

Despite the fact that change does not follow the pattern of baseline smooth growth of Lehman and the fact that change obeys a Zipfian distribution with a peak at zero, we have to say that the presence of feedback in the evolution process is evident; thus the law holds.

4.2 Properties of Growth for Schema Evolution

Law of continuing growth (Law VI). The sixth law of continuing growth requires us to verify whether the information capacity of the system (schema size) continuously grows. In all occasions, the schema size increases in the long run (Fig. 1, 2). We frequently observe some shrinking events in the timeline of schema growth in all data sets. However, *all data sets demonstrate the tendency to grow over time*. However, we also have differences from the traditional software systems that the law studies: as with Law I, the term ”continually” is questionable. As already mentioned (refer to Law III and Fig. 1, 2), change comes with frequent (and sometimes long) periods of *stability*, where the size of the schema does not change (or changes very little).

Therefore we can conclude that *the law holds, albeit modified to accommodate the particularities of database schemata.*

Law of conservation of familiarity (Law V). A first question, of central interest for the fifth law's intuition is: “What happens after excessive changes? Do we observe small ripples of change, showing the absorbing of the change's impact in terms of corrective maintenance and developer acquaintance with the new version of the schema?” An accompanying question, typically encountered in the literature, is: “What is the effect of age over the growth and the growth ratio of the schema?” Is it slowly declining, constant or oblivious to age? Again, we would like to remind the reader on the properties of growth, discussed in Law III of self-regulation: the changes are small, come with spike patterns between zero and non-zero deltas and the average value of growth is very close to zero.

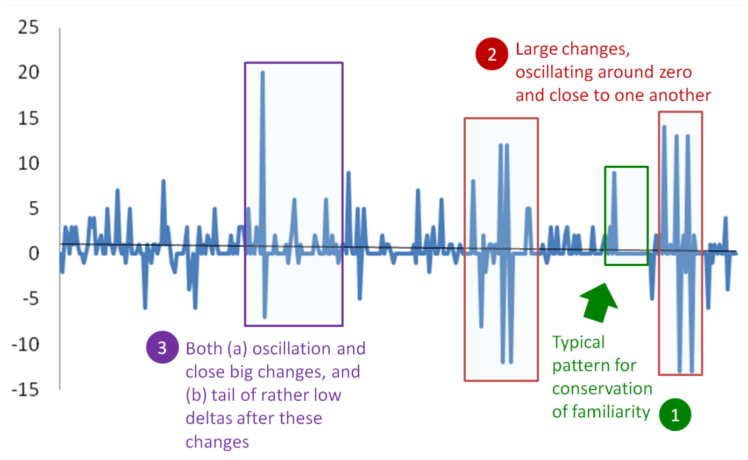


Fig. 6. Different patterns of change in attribute growth of Mediawiki (over version-id, concealed for fig. clarity)

Concerning the ripples after large changes, we can detect several patterns. Observe Fig. 6, depicting attribute growth for the MediaWiki dataset. Due to the fact that this involves the growth of attributes, the phenomena are amplified compared to the case of tables. Reading from right to left, we can see that there are indeed cases where a large spike is followed by small or no changes (case 1). However, within the small pool of large changes that exist overall, it is quite frequent to see sequences of large oscillations one after the other, and quite frequently being performed around zero too (case 2). In some occurrences, we see both (case 3).

Concerning the effect of age, we do not see a diminishing trend in the values of growth; however, *age results to a reduction in the density of changes and the frequency of non-zero values in the spikes. This explains the drop of the average value in almost all the studied data sets* (Fig. 4): the linear interpolation drops; however, this is not due to the decrease of the height of the spikes, but due to the decrease of their density.

The heartbeat of the systems tells a similar story: typically, change is quite more frequent in the beginning, despite the fact that existence of large changes and dense periods of activities can occur in any period of the lifetime. Fig. 1 and 2 clearly demonstrate this by combining schema size and activity. This trend is typical for almost all of the studied databases (with phpBB being the only exception, demonstrating increased activity in its latest versions with the schema size oscillating between 60 and 63 tables).

Concerning the validity of the law, we *believe that the law is possible but not confirmed*. The law states that the growth is constrained by the need to maintain familiarity. However, the peculiarity of databases, compared to typical software systems, is that there are other good reasons to constrain growth: (a) a high degree of dependence of other modules from the database, and, (b) an intense effort to make the database clean and organized. Therefore, conservation of familiarity, although important cannot solely justify the limited growth. The extent of the contribution of each reason is unclear.

Law of conservation of organizational stability (Law IV). To validate the hypothesis that the law of conservation of organizational stability holds, we need to establish that the project's lifetime is divided in phases, each of which (a) demonstrates a constant growth, and, (b) is connected to the next phase with an abrupt change. Moreover, abrupt changes should occur from time to time and not all the time (resulting in extremely short phases).

If we focus on the essence of the law, we can safely say that it does not hold. The heartbeats of Fig. 1 and 2 and the arbitrary sequencing of spikes and stability (Fig. 4, 6) make it impossible to speak about constant growth, even in phases. The open-source nature of our cases plays a role to that too.

4.3 Perfective Maintenance for Schema Evolution

Law of increasing complexity (Law II). The law states that complexity increases with age, unless effort is taken to prevent this – nevertheless, the law does not prescribe a clear assessment method for its validity. The rationale behind verifying the law dictates the observation of (a) an increasing trend in complexity of a software system, battled by (b) a perfective maintenance activity that attempts to reduce it and demonstrated by drops in the system size and rate of expansion. As there is no precise definition and measurement of complexity in the law, different metrics have been employed (coupling, cyclomatic complexity, etc. – see [6] for a review). Unfortunately, as most of these metrics are non-applicable to the case of databases, we take a definition already found in Lehman [1]: *complexity is defined as the number of modules handled (in our case tables added or modified) over the absolute value of growth per transition.* This formula approximates how much effort has been invested in expanding the system over the actual difference achieved (large values demonstrate too much effort for too small change).

Related literature typically speaks for increasing complexity [1], [2], [3], [6], although there have been counterarguments for the case of open source software [5]. In our case, *in all the datasets but Biosql, complexity, as defined in the previous paragraph, does not increase* (Fig. 7). The phenomenon must be coupled with the drop in change density (Law V) and although we cannot provide undisputable explanation, we offer the synergy of two causes: (a) the increasing dependence of the surrounding code to the database that makes developers more cautious to perform schema changes as they incur higher maintenance costs, and, (b) the success of the perfective maintenance, which results in a clean schema, requiring less corrective maintenance in the future.

Although we cannot confirm or disprove the law based on undisputed objective measurements, we have indications that the second law partially holds, albeit with *completely different connotations than the ones reported by Lehman for typical software systems: in the case of database schemata, complexity, when measured as the fraction of expansion effort over actual growth, drops.*

Law of declining quality (Law VII) The seventh law postulates that quality declines with age unless the system is rigorously adapted to its external environment. Lehman and Fernandez-Ramil [3] avoid both (a) a definition of quality "the definition, measurement, modelling and monitoring of software quality-related characteristics are very dependent on application, organisation, product and process characteristics and goals", and, (b) giving any other support to the law than a logical proof: as the system expands over time, its complexity rises and thus the addressing of user requirements and removal of defects becomes more and more difficult, unless work is done to confront the phenomenon ("the decline in software quality with age, appears to relate to a growth in complexity that must be associated with ageing").

We have already demonstrated that the rationale behind complexity increase is not supported by our observations. At the same time, we cannot assess schema quality with undisputed means. Therefore, we cannot confirm or disprove the law based on undisputed objective measurements.

5 Discussion

In this section, we summarize fundamental *observations* and *patterns* that have been detected in our study. We intentionally avoid the term *law*, as we do not have unshakeable evidence for their explanation. Apart from the *empirical grounding*, due a very large amount of datasets that obey the same patterns (which we believe we have fairly attained), we would require an undisputed *rationalized grounding*, that can be obtained via a clear explanation of the underlying mechanism that guides them, also established on measured, undisputed data.

Feedback-based Behavior for Schema Evolution. *As an overall trend, the information capacity of the database schema is enhanced – i.e., the size*

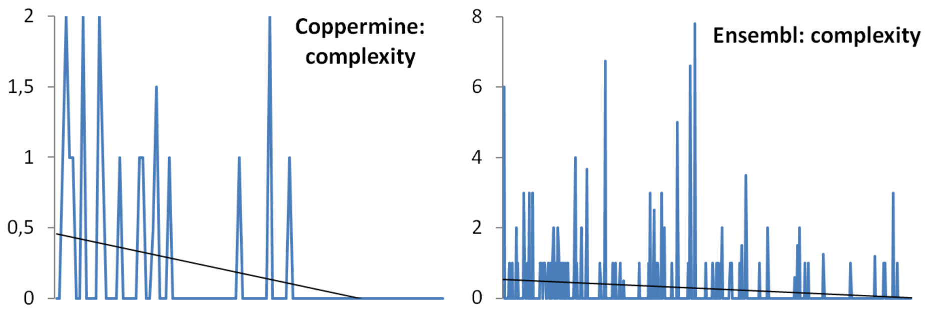


Fig. 7. Complexity for Coppermine and Ensembl (over version-id, concealed for clarity)

grows in the long term (VI). The existence of perfective maintenance is evident in almost all datasets with the existence of relation and attributes removals, as well as observable drops in growth and size of the schema (sometimes large ones). In fact, growth frequently oscillates between positive and negative values (III). The schema size of a certain version of the database can be accurately estimated via a regressive formula that exploits the amount of changes in recent, previous versions (VIII). Based on the above, we can state that the essence of Lehman's laws applies to open-source databases too: *Schema evolution demonstrates the behavior of a feedback-regulated system, as it obeys the antagonism between the need for expanding its information capacity to address user needs and the need to control the unordered expansion, with perfective maintenance.*

Observations concerning the heartbeat of change. *The database is not continuously adapted, but rather, alterations occur from time to time, both in terms of versions and in terms of time (I). Change does not follow patterns of constant behaviour (IV). Age results in a reduction of the density of changes to the database schema in most cases (V).*

Schema growth is small (observations): *Growth is typically small in the evolution of database schemata, compared to traditional software systems (III). The distribution of occurrences of the amount of schema change follows a Zipfian distribution, with a predominant amount of zero growth in all data sets. Plainly put, there is a very large amount of versions with zero growth, both in the case of attributes and in the case of tables. The rest of the frequently occurring values are close to zero, too. The average value of growth is typically close to zero (although positive) (III) and drops with time, mainly due to the drop in change density (V).*

Threats to validity. We start with a *fundamental inquiry*: are databases E-type systems, so that this research is meaningful in the first place? Despite a fundamental difference (as databases involve information and not functional capacity), databases, closely resemble E-type systems as they address the real problem of query answering, come with their own user community (developers, DBA's), and act as fairly independent modules in information systems. Concerning the *external validity* of our study, its context concerns *the study of the evolution of the $\underline{\text{logical}}$ schema of databases in $\underline{\text{open-source}}$ software.* We avoid generalizing our findings to databases operating in closed environments and we stress that our study has focused only on the logical structure of databases, avoiding physical properties (let alone instance-level observations). Overall, we believe we have provided a safe, representative experiment with a significant number of schemata, having different purposes in the real world and time span (from rather few (40) to numerous (500+) versions). Our findings are generally consistent (with few exceptions that we mentioned). Concerning *internal validity* and cause-effect relationships, we avoid directly relating age with phenomena like the dropping density of changes or the size growth; on the contrary, we attribute the phenomena to a confounding variable, perfective maintenance actions, which we anticipate to be causing the observed behavior. When it comes to *construct validity*, all the measures we have employed are accurate, consistent with the metrics used in the related literature and appropriate for assessing the law to which they are employed. The only exceptions to this statement are Laws II and VII dealing with the complexity and the quality of the schemata. Both terms are very general and the related database literature does not really provide adequate metrics other than size-related (which we deem too simple for our purpose); our own measurement of complexity requires deeper investigation. Therefore, the undisputed assessment of these laws remains open.

Future Work. The extension of the study to more datasets, possibly non-relational too, and the study of databases in closed environments for large periods of time, are possible roads for future research. Concerning the current findings of our study, the detailed understanding of the feedback mechanism, especially when it comes to ageing and complexity (Law II) as well as patterns of growth (Laws III and V), or patterns in the heartbeat of the evolution, are open issues worth investigating.

Acknowledgment. This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

1. Belady, L.A., Lehman, M.M.: A model of large program development. IBM Systems Journal **15**(3) (1976) 225–252
2. Lehman, M.M., Ramil, J.F., Wernick, P., Perry, D.E., Turski, W.M.: Metrics and laws of software evolution - the nineties view. In: 4th IEEE International Software Metrics Symposium (METRICS 1997). (1997) 20
3. Lehman, M.M., Fernandez-Ramil, J.C.: Rules and Tools for Software Evolution Planning and Management. In: Software Evolution and Feedback: Theory and Practice. John Wiley and Sons Ltd (2006) ISBN-13: 978-0-470-87180-5.
4. Xing, Z., Stroulia, E.: Analyzing the evolutionary history of the logical design of object-oriented software. IEEE Trans. Software Eng. **31**(10) (2005) 850–868
5. Fernández-Ramil, J., Lozano, A., Wermelinger, M., Capiluppi, A.: Empirical studies of open source evolution. In: Software Evolution. (2008) 263–288
6. Xie, G., Chen, J., Neamtiu, I.: Towards a better understanding of software evolution: An empirical study on open source software. In: 25th IEEE International Conference on Software Maintenance (ICSM 2009), Edmonton, Alberta, Canada. (2009) 51–60
7. Sjøberg, D.: Quantifying schema evolution. Information and Software Technology **35**(1) (1993) 35–44
8. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: Metrics for the prediction of evolution impact in etl ecosystems: A case study. J. Data Semantics **1**(2) (2012) 75–97
9. Curino, C., Moon, H.J., Tanca, L., Zaniolo, C.: Schema evolution in wikipedia: toward a web information system benchmark. In: Proceedings of ICEIS 2008, Citeseer (2008)
10. Curino, C.A., Moon, H.J., Zaniolo, C.: Graceful database schema evolution: the prism workbench. Proceedings of the VLDB Endowment **1** (2008) 761–772

Information Systems 53 (2015) 363–385

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Growing up with stability: How open-source relational databases evolve

Ioannis Skoulis[a,1], Panos Vassiliadis[b,*], Apostolos V. Zarras[b]

[a] Opera, Helsinki, Finland
[b] University of Ioannina, Ioannina, Greece

ARTICLE INFO

Available online 30 April 2015

Keywords:
Schema evolution
Software evolution
Lehman's laws

ABSTRACT

Like all software systems, databases are subject to evolution as time passes. The impact of this evolution can be vast as a change to the schema of a database can affect the syntactic correctness and the semantic validity of all the surrounding applications. In this paper, we have performed a thorough, large-scale study on the evolution of databases that are part of larger open source projects, publicly available through open source repositories. Lehman's laws of software evolution, a well-established set of observations on how the typical software systems evolve (matured during the last forty years), has served as our guide towards providing insights on the mechanisms that govern schema evolution. Much like software systems, we found that schemata expand over time, under a stabilization mechanism that constraints uncontrolled expansion with perfective maintenance. At the same time, unlike typical software systems, the growth is typically low, with long periods of calmness interrupted by bursts of maintenance and a surprising lack of complexity increase.

*A truly stable system expects the unexpected, is prepared to be disrupted, waits to be transformed.*Tom Robbins, Even Cowgirls Get the Blues

1. Introduction

Software evolution is the change of a software system over time, typically performed via a remarkably difficult, complicated and time consuming process, software maintenance. Schema evolution is the most important aspect of software evolution that pertains to databases, as it can have a tremendous impact to the entire information system built around the evolving database, severely affecting both developers and end-users. Quite frequently, development waits till a "schema backbone" is stable and applications are build on top of it. This is due to the "*dependency magnet*" nature of databases: a change in the schema of a database may immediately drive surrounding applications to crash (in case of deletions or renamings) or be semantically defective or inaccurate (in the case of information addition, or restructuring). Therefore, discovering laws, patterns and regularities in schema evolution can result in great benefits, as we would be able to design databases with a view to their evolution and minimize the impact of evolution to the surrounding applications: (a) by avoiding "design anti-patterns" leading to cumulative complexity for both the database and the surrounding applications and (b) by planning administration and maintenance tasks and resources, instead of just responding to emergencies.

In sharp distinction to traditional software systems, and disproportionately to the severity of its implications, database evolution has hardly been studied throughout the entire lifetime of the data management discipline. It is only amazing

* Corresponding author.
E-mail addresses: giskou@gmail.com (I. Skoulis), pvassil@cs.uoi.gr (P. Vassiliadis), zarras@cs.uoi.gr (A.V. Zarras).
[1] Work conducted while in the University of Ioannina.

http://dx.doi.org/10.1016/j.is.2015.03.009
0306-4379/© 2015 Elsevier Ltd. All rights reserved.

to find out that, in the history of the discipline, just a handful of studies had been published in the area. The deficit is really amazing in the case of traditional database environments, where only two(!) studies [1,2] have been published. Apart from amazing, this deficit should also be expected: allowing the monitoring, study and eventual publication of the evolution properties of a database would expose the internals of a critical part of the core of an organization's information system. Fortunately, the open-source movement has provided us with the possibility to slightly change this landscape. As public repositories (git, svn, etc.) keep the entire history of revisions of software projects, including the schema files of any database internally hosted within them, we are now presented with the opportunity to study the version histories of such open source databases. Hence, within only a few years in the late ‘00's, several research efforts [3–6] have studied of schema evolution in open source environments. Those studies, however, focus on the statistical properties of the evolution and do not provide details on the mechanism that governs the evolution of database schemata.

To contribute towards amending this deficit, *the research goal of this paper involves the identification of patterns and regularities of schema evolution that can help us understand the underlying mechanism that governs it*. To this end, we study the evolution of the logical schema of eight databases, that are parts of publicly available, open-source software projects (Section 3). We have collected and cleansed the available versions of the database schemata for the eight case studies, extracted the changes that have been performed in these versions and, finally, we have come up with usable datasets that we subsequently analyzed.

Our main tool for this analysis came from the area of software engineering. In an attempt to understand the mechanics behind the evolution of software and facilitate a smoother, lest disruptive maintenance process, Meir Lehman and his colleagues introduced a set of rules in mid seventies [7], also known as the *Laws on Software Evolution* (Section 2). Their findings, that were reviewed and enhanced for nearly 40 years [8,9], have, since then, given an insight to managers, software developers and researchers, as to *what* evolves in the lifetime of a software system, and *why* it does so. Other studies (see [10] for a survey) have complemented these insights in this field, typically with particular focus to open-source software projects. In our case, we adapted the laws of software evolution to the case of schema evolution and utilized them as a driver towards understanding how the studied schemata evolve. Our findings (Section 4) indicate that the schemata of open source databases expand over time, with long periods of calmness connected via bursts of maintenance effort focused in time, and with significant effort towards the perfective maintenance of the schema that appears to result in an unexpected lack of complexity increase. Incremental growth of the schema is typically low and its volume follows a Zipfian distribution. In both the presentations of our results and in our concluding notes (Section 5) we also demonstrate that although the technical assessment of Lehman's laws shows that the typical software systems evolve quite differently than database schemata, the essence of the laws is preserved: evolution is not about uncontrolled expansion; on the contrary, there appears to be a stabilization mechanism that employs perfective maintenance to control the otherwise growing trend of increase in the information capacity of the database.

Roadmap: In Section 2, we summarize Lehman's laws for the non-expert reader and survey related efforts, too. In Section 3 we discuss the experimental setup of this study and in Section 4, we detail our findings. We conclude our deliberations with a summary of our findings and their implications in Section 5.

2. Lehman laws of software evolution in a nutshell

Meir M. Lehman and his colleagues, have introduced, and subsequently amended, enriched, and corrected a set of rules on the behavior of software as it evolves over time [7–9]. Lehman's laws focus on *E-type systems* that concern “software solving a problem or addressing an application in the real-world” [8]. The main idea behind the laws of evolution for E-type software systems is that their *evolution is a process that follows the behavior of a feedback-based system*. Being a feedback-based system, the evolution process has to balance (a) *positive feedback*, i.e., the need to adapt to a changing environment and grow to address the need for more functionality, and, (b) *negative feedback*, i.e., the need to control, constrain and direct change in ways that prevent the deterioration of the maintainability and manageability of the software. In the sequel, we list the definitions of the laws as they are presented in [9], in a more abstract form than previous versions and with the benefit of retrospect, after thirty years of maturity and research findings.

(I) *Law of Continuing Change*: An E-type system must be continually adapted or else it becomes progressively less satisfactory in use.

(II) *Law of Increasing Complexity*: As an E-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.

(III) *Law of Self-regulation*: Global E-type system evolution is feedback regulated.

(IV) *Law of Conservation of Organizational Stability*: The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime.

(V) *Law of Conservation of Familiarity*: In general, the incremental growth (growth ratio trend) of E-type systems is constrained by the need to maintain familiarity.

(VI) *Law of Continuing Growth*: The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.

(VII) *Law of Declining Quality*: Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining.

(VIII) *Law of Feedback System*: E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

Before proceeding with our study, we present a first apodosis of the laws, taking into consideration both the wording of the laws, but most importantly their accompanying explanations [9].

An E-Type software system continuously changes over time (I) obeying a complex feedback-based evolution process (VIII). On the one hand, due to the need for growth and adaptation that acts as positive feedback, this process results in an increasing functional capacity of the system (VI), produced by a growth ratio that is slowly declining in the long term (V). The process is typically guided by a pattern of growth that demonstrates its self-regulating nature: growth advances smoothly; still, whenever there are excessive deviations from the typical, baseline rate of growth (either in a single release, or accumulated over time), the evolution process obeys the need for calibrating releases of perfective maintenance, i.e., code restructuring and documentation for better maintainability and comprehension (expressed via minor growth and demonstrating negative feedback) to stop the unordered growth of the system's complexity (III). On the other hand, to regulate the ever-increasing growth, there is negative feedback in the system controlling both the overall quality of the system (VII), with particular emphasis to its internal quality (II). The effort consumed for the above process is typically constant over phases, with the phases disrupted with bursts of effort from time to time (IV).

2.1. Lehman's laws and related empirical studies

Software evolution is an active research field for more than 40 years and concerns different levels of abstraction, including the software architecture [11], design [12] and implementation [10]. Lehman's theory of software evolution is the cornerstone of the efforts that have been performed all these years. For a detailed historical survey on the evolution of Lehman's theory and other related works the interested reader can refer to [10]. Following, we briefly discuss the milestones and key findings that resulted from these efforts.

Lehman's theory of software evolution was first introduced in the 70s. Back then, the theory included the first three laws, concerning the continuous change, the increasing complexity and the self-regulating properties of the software evolution process [7]. The experimental evidence that produced these laws was based on a single case study, namely the OS/360 operating system. During the 70s and the 80s the formulation of the first three laws has been revised, with respect to further results and empirical observations that came up [13]. Moreover, Lehman's theory has been extended with the fourth and the fifth law that concerned the issues of organizational stability and conservation of familiarity [13]. In the 90s, based on additional case studies, the laws have been revised again and extended with the last three laws, referring to the continuous growth, the declining quality and to the feedback mechanism that governs the evolution process [14,8,15]. Lehman's theory did not grow since then, the set of laws has been stabilized, and most of the activity around them concerned moderate changes in their formulation, performed in the 00s [9].

During all these years there have also been studies by other authors on the validity of the laws [16,17]. An interesting finding uncovered from these efforts is that the behavior of commercial software differs from that of academic and research software, with the former kind being much more faithful to the laws, compared to the latter two kinds. The partial validity of the laws is also highlighted in [18], along with the need for a more formal framework that would facilitate the assessment of the laws.

The diverse behavior of software concerning the validity of Lehman's laws is emphasized in subsequent studies that investigated the evolution of open source software. Most of these studies found only partial support for the validity of the laws. The efforts in this line of research vary from the pioneer studies of Godfrey and Tu [19,20], focusing mainly on Linux, to large scale studies [21–24]. The common ground in all these studies is that they found support for the laws of continuing change and growth. Refs. [23,25] concluded in the validation of more laws, including the ones of self-regulation and conservation of familiarity. Moreover, [26] revealed that the laws may be valid after a certain point in the software lifecycle. In particular, taking a step further from the efforts of Godfrey and Tu, [26] found that after a certain version the evolution of Linux follows, at least partially, most of the laws.

2.2. Empirical studies on database evolution

Being at the very core of most software, databases are also subject to evolution, which concerns changes in their contents and, most importantly, their schemas. Database evolution can concern (a) changes in the operational environment of the database, (b) changes in the content of the databases as time passes by, and (c) changes in the internal structure, or *schema*, of the database. *Schema evolution*, itself, can be addressed at (a) the conceptual level, where the understanding of the problem domain and its representation via an ER schema evolves, (b) the logical level, where the main constructs of the database structure evolve (for example, relations and views in the relational area, classes in the object-oriented database area, or (XML) elements in the XML/semi-structured area), and (c) the physical level, involving data placement and partitioning, indexing, compression, archiving, etc.

Interestingly, *the related literature on the actual mechanics of schema evolution includes only a few case studies*, as the research community would find it very hard to obtain access to monitor database schemata for an in depth study over a significant period of time. Despite the fact that in our work we study *schema evolution at the logical level of databases in open-source software*, here, we proceed to survey all the works we are aware about in the broader area of schema evolution.

The first paper [1] discusses the evolution of the database of a health management system over a period of 18 months, monitored by a tool specifically constructed for this purpose. A single database schema was examined, and the monitoring revealed that all the tables of the schema were affected and the schema had a 139% increase in size for relations and 274% for attributes. The consequences of this evolution were significantly large as a cumulative 45% of all the names that were used in the queries had to be deleted or inserted.

Fifteen years later, the authors of [3] made an analysis on the database back-end of MediaWiki, the software that powers Wikipedia. The study conducted over the versions of four years, revealed a 100% increase in schema size, the observation that around 45% of changes do not affect the information capacity of the schema (but are rather index adjustments, documentation, etc.), and a statistical study of lifetimes, change breakdown and version commits. Special mention should be made to this line of research [27], as it is based on PRISM (recently re-engineered to PRISM++ [28]), a change management tool that provides a language of Schema Modification Operations (SMO) (that model the creation, renaming and deletion of tables and attributes, and their merging and partitioning) to express schema changes. More importantly, the people involved in this line of research should be credited for providing a large collection of links[2] for open source projects that include database support.

A work in the area of data warehousing [2] monitored the evolution of seven ETL scripts along with the evolution of the source data. The experimental analysis of the authors is based in a six-month monitoring of seven real-world ETL scenarios that process data for statistical surveys. The findings of the study indicate that schema size and module complexity are important factors for the vulnerability of an ETL flow to changes. This work has been part of an effort to provide what-if analysis facilities to the management of schema evolution via the Hecataeus tool (see [29,30]).

Finally, certain efforts studied the evolution of databases, while taking into account the applications that use them. In particular, in [5] the authors considered 4 case studies of embedded databases (i.e., databases tightly coupled with corresponding applications that rely on them) and studied the different kinds of changes that occurred in these cases. Moreover, they performed a respective frequency and timing analysis, which showed that the database schemas tend to stabilize over time. In [4], the authors focused on two case studies. The results of this effort revealed that database schema changes and that the source code of dependent applications does not always evolve in sync with changes to the database schema. Ref. [4] further provides a discussion concerning that the impact of database schema changes on the application code. Ref. [6] takes a step further with an empirical study of the co-evolution of database schemas and code. This effort investigated ten case studies. The results indicate that database schemas evolve frequently during the application lifecycle, with schema changes implying a significant amount of code level modifications.

2.3. Novelty with respect to the state of the art

Going beyond the related literature on software evolution, in general, and database evolution, in particular, our CAiSE'14 paper [31] investigated for the first time patterns and regularities of database evolution, based on Lehman's laws. To this end, we conducted a large scale case study of eight databases, that are parts open-source software projects. This paper extends our prior work with further details concerning the intuition and the relevance of the laws in the case of databases, the metrics that have been used in the literature for the assessment of the laws, and the metrics that we employed in the case of databases. More importantly, we provide detailed presentations of the results and thorough discussions of our findings.

3. Experimental setup of the study

Datasets: We have studied eight database schemata from open-source software projects. Fig. 1 lists the datasets along with some interesting properties.

ATLAS[3] is a particle physics experiment at the Large Hadron Collider at CERN, Geneva, Switzerland, with the goal of learning about the basic forces that have shaped our universe. ATLAS Trigger is the software responsible for filtering the immense data (40 TB per second) collected by the Collider and storing them in its Oracle database.

BioSQL[4] is a generic relational model covering sequences, features, sequence and feature annotation, a reference taxonomy, and ontologies (or controlled vocabularies) from various sources such as GenBank or Swissport. While originally conceived as a local relational store for GenBank, the project has since become a collaboration platform between the Open Bioinformatics Foundation (OBF) projects (including BioPerl, BioPython, BioJava, and BioRuby). The goal is to build a sufficiently generic schema for persistent storage of sequences, features, and annotation in a way that is interoperable between these Bio* projects.

Ensembl is a joint scientific project between the European Bioinformatics Institute (EBI)[5] and the Wellcome Trust Sanger Institute (WTSI)[6] which was launched in 1999 in response to the imminent completion of the Human Genome Project. The goal of Ensembl was to automatically annotate the three billion base pairs of sequences of the genome, integrate this annotation with other available biological data and make all this publicly available via the web. Since the launch of the website, many more genomes have been added to Ensembl and the range of available data has also expanded to include comparative genomics, variation and regulatory data.

MediaWiki[7] was first introduced in early 2002 by the Wikimedia Foundation along with Wikipedia, and hosts Wikipedia's content since then. As an open source system (licensed under the GNU GPL) written in PHP, it was also adopted by many companies and is used in thousands of websites both as a knowledge management system, and for collaborative group projects.

[2] http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Benchmark_Extension

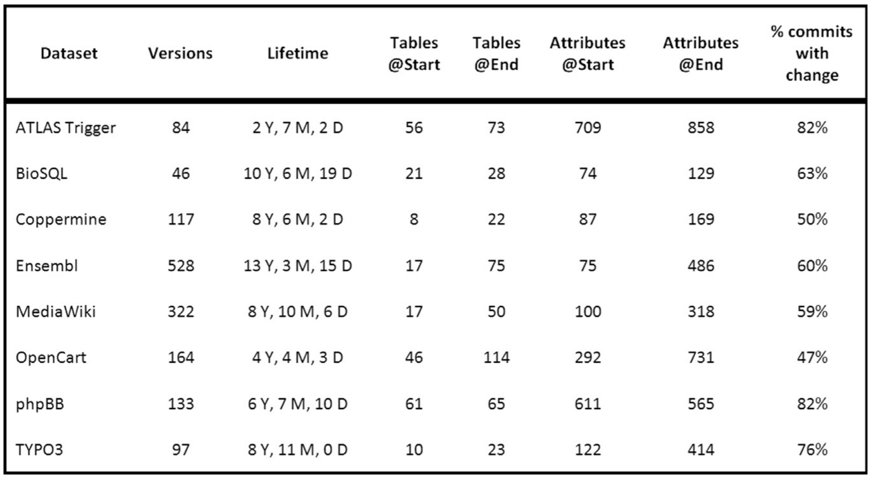
[3] http://atlas.web.cern.ch/Atlas/Collaboration/

[4] http://www.biosql.org/wiki/Main_Page

[5] https://www.ebi.ac.uk/

[6] https://www.sanger.ac.uk/

[7] https://www.mediawiki.org/wiki/MediaWiki



Dataset	Versions	Lifetime	Tables @Start	Tables @End	Attributes @Start	Attributes @End	% commits with change
ATLAS Trigger	84	2 Y, 7 M, 2 D	56	73	709	858	82%
BioSQL	46	10 Y, 6 M, 19 D	21	28	74	129	63%
Coppermine	117	8 Y, 6 M, 2 D	8	22	87	169	50%
Ensembl	528	13 Y, 3 M, 15 D	17	75	75	486	60%
MediaWiki	322	8 Y, 10 M, 6 D	17	50	100	318	59%
OpenCart	164	4 Y, 4 M, 3 D	46	114	292	731	47%
phpBB	133	6 Y, 7 M, 10 D	61	65	611	565	82%
TYPO3	97	8 Y, 11 M, 0 D	10	23	122	414	76%

Fig. 1. The datasets employed in our study.

Coppermine[8] is a photo gallery web application. OpenCart[9] is an open source shopping cart system. PhpBB[10] (PHP Bulletin Board) is an Internet forum package written in PHP. TYPO3[11] is a web content management framework based on PHP. All these platforms are highly rated and used.

Dataset Collection and Processing: A first collection of links to available datasets was made by the authors of [3,27][12]; for this, these authors deserve honorable credit. We isolated eight databases that appeared to be alive and used (as already mentioned, some of them are actually quite prominent). For each dataset, we have gathered the schema versions (DDL files) that were available at June 2013, directly from public source code repositories (cvs, svn, git) for the eight datasets listed in Fig. 1. We have targeted main development branches and trunks to maximize the validity of the gathered resources. *We are interested only on changes of the database part of the project as they are integrated in the trunk of the project*. Hence, we collected all the commits of the trunk or master branch that were available at the time, and ignored all other branches of the project, as well as any commits of other modules of the project that did not affect the database.

For all of the projects, we focused on their release for MySQL (except ATLAS Trigger, available only for Oracle). Those files were then renamed with their filenames matching to the date (in standard UNIX time) the commit was made. The files were then processed in sequential pairs from our tool, Hecate, to give us in a fully automated way (a) the differences between two subsequent commits and (b) the measures we needed to conduct this study. Attributes are marked as altered if they exist in both versions and their type or participation in their tables's primary key changed. Tables are marked as altered if they exist in both versions and their contents have changed (attributes inserted/deleted/altered).

[8] http://coppermine-gallery.net/
[9] http://www.opencart.com
[10] https://www.phpbb.com/
[11] http://typo3.org/
[12] http://data.schemaevolution.org

All the datasets used, along with our tool-suite for managing the evolution of databases can be found in our group's git: https://github.com/DAINTINESS-Group.

4. Assessing the laws for schema evolution

The laws of software evolution where developed and reshaped over forty years. Explaining each law in isolation from the others is precarious, as it risks losing the deeper essence and inter-dependencies of the laws [9]. To this end, in this section, we organize the laws in three thematic areas of the overall evolution management mechanism that they reveal. The first group of laws discusses the existence of a feedback mechanism that constrains the uncontrolled evolution of software. The second group discusses the properties of the growth part of the system, i.e., the part of the evolution mechanism that accounts for positive feedback. The third group of laws discusses the properties of perfective maintenance that constrains the uncontrolled growth, i.e., the part of the evolution mechanism that accounts for negative feedback. To quantitatively support our study, we utilize the following measures:

- *Schema size of a version*: The number of tables of a schema version.
- *Schema Growth*: The difference between the schema size of two (typically subsequent) versions (i.e., new–old).
- *Heartbeat*: A sequence of tuples, one per transition, with the count of the events that occurred during this transition. In the context of this paper, for each transition between two subsequent versions, we produce a tuple of measures including Table Insertions, Table Deletions, Attribute Insertions, Attribute Deletions, Attribute Alternations (change of data type), Attributes Inserted at Table Formation, Attribute Deletions at Table Removal. To clarify, Attribute Insertions concern additions of attributes to an existing table, whereas Attributes Inserted at Table Formation concern the number of attributes generated whenever a new table

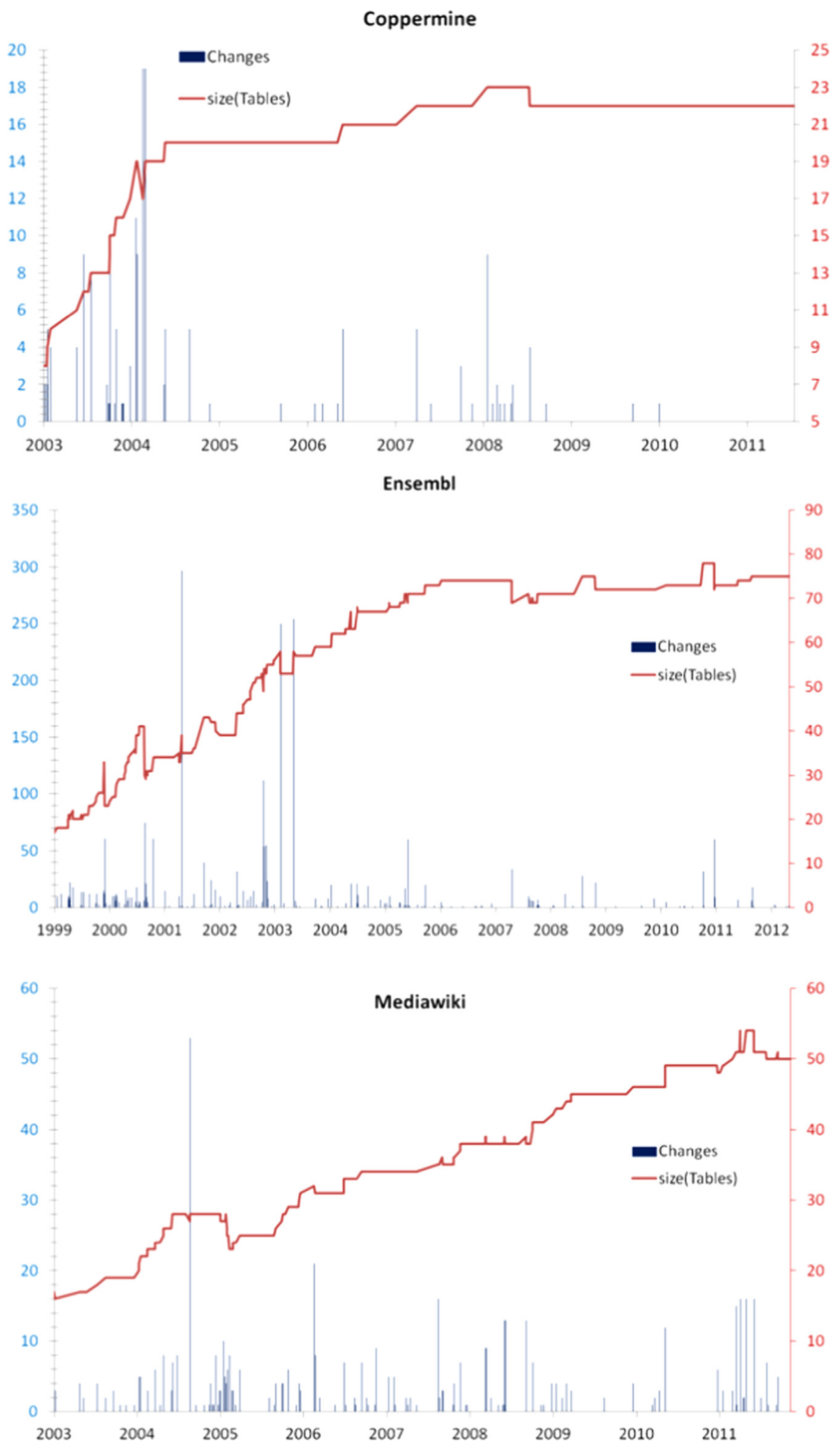


Fig. 2. Combined demonstration of heartbeat (number of changes per version) and schema size (no. of tables). The left axis signifies the amount of change and the right axis the number of tables.

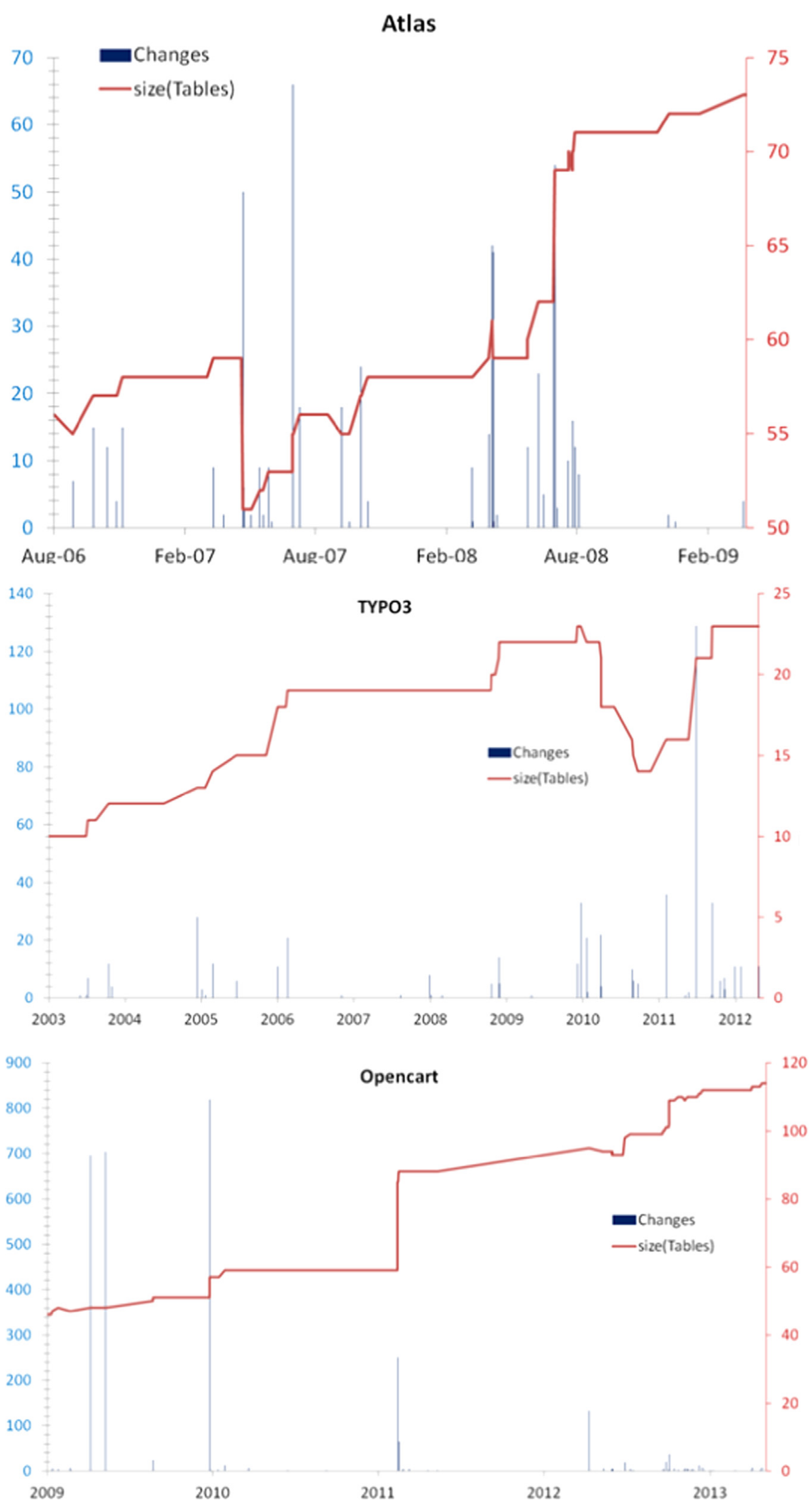


Fig. 3. Combined demonstration of heartbeat (continued).

is born. Attribute Deletions concern deletions from a table that continues to exist, whereas Attribute Deletions at Table Removal concern attributes that are removed whenever their containing table is removed. We sum up these measures per transition, to produce the Heartbeat of the lifetime of the dataset.

We would like to remind the reader that we study the *evolution of the logical schema of databases in open-source software*. In all our deliberations, we take the above context as granted and avoid repeating it for reasons of better presentation of our results.

4.1. Is there a feedback-based system for schema evolution?

4.1.1. Law of continuing change (Law I)

The first law argues that the system continuously changes over time.

An E-type system must be continually adapted or else it becomes progressively less satisfactory in use.

The main idea behind this law is simple: as the real world environment evolves, the software that is intended to address its problems has to evolve too. If this does not happen, the system becomes less satisfactory.

Metrics for the assessment of the law's validity: To establish the law, one needs to show that the software shows signs of evolution as time passes. Possible metrics from the field of software engineering [23] include (a) the cumulative number of changes and (b) the breakdown of changes over time.

Assessment: To validate the hypothesis that the law of continuing change holds, we study the heartbeat of the schema's life (see Figs. 2–4 for a combined demonstration of heartbeat and schema size).

With the exception of BioSQL that appeared to be "sleeping" for some years and was later re-activated, in all other cases, we have changes (sometimes moderate, sometimes

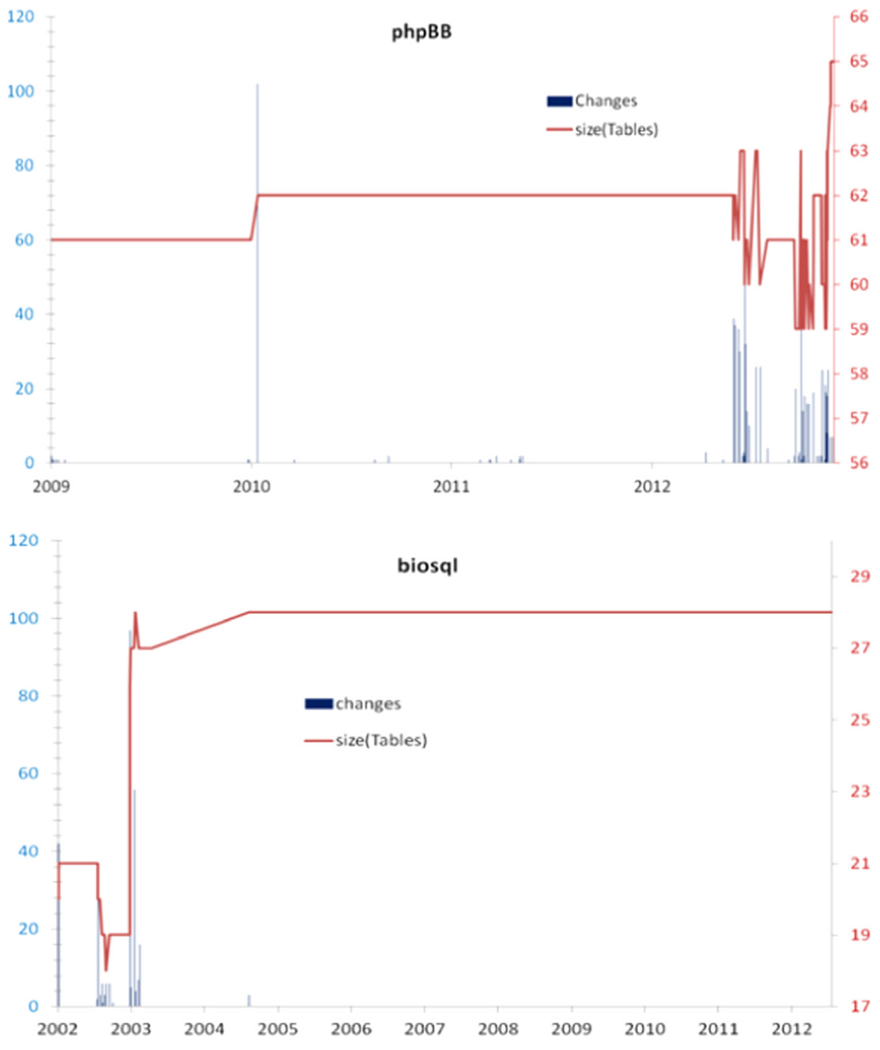


Fig. 4. Combined demonstration of heartbeat (continued).

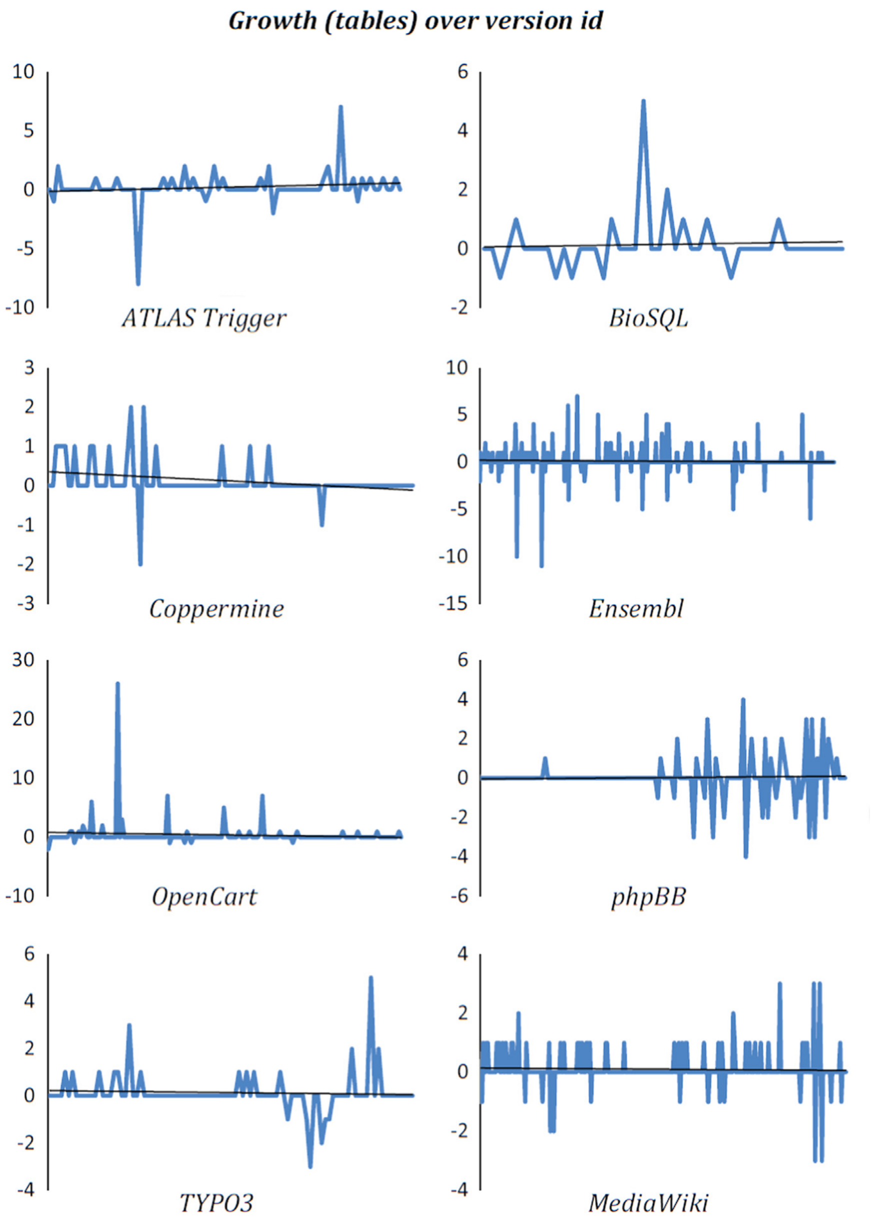


Fig. 5. Growth (tables) over version id for all the datasets.

even excessive) over the entire lifetime of the database schema. An important observation stemming from the visual inspection of our change-over-time data, is that the term *continually* in the law's definition is challenged: *we observe that database schema evolution happens in bursts, in grouped periods of evolutionary activity, and not as a continuous process*! Take into account that the versions with zero changes are versions where either commenting and beautification takes place, or the changes do not refer to the information capacity of the schema (relations, attributes and constraints) but rather, they concern the physical level properties (indexes, storage engines, etc.) that pertain to performance aspects of the database.

Can we state that this stillness makes the schema "unsatisfactory" (referring back to the wording of the first law by Lehman)? We believe that the answer to the question is negative: since the system hosting the database continues to be in use, user dissatisfaction would actually call for continuous growth of the database, or eventual rejection of the system. This does not happen. On the other hand, our explanation relies on the reference nature of the database in terms of software architecture: if the database evolves, the rest of the code, which is basically using the database (and not vice versa), breaks.

Overall, if we account for the exact wording of the law, we conclude that the law partially holds.

4.1.2. Law of Self-regulation (Law III)

The third law of software evolution is known as the law of "Self-regulation" and comes with a laconic definition.

Global E-type system evolution is feedback regulated.

The main idea behind this law is that the system under development is actually a feedback-regulated system: development and maintenance take place and there is positive and negative feedback to the system. As the clients of the system request more functionality, the system grows in size to address this demand; at the same time, as the system grows, corrective and perfective maintenance has to take place to remove bugs and improve the internal quality of the software (reduced complexity, increased understandability) [8].

Thus, the system's growth cannot continually evolve with the same rate; on the contrary, what one expects is to see a typical "baseline" growth, interrupted with releases of perfective maintenance. This trend is so strong, that, in the long run, the system's size demonstrates what the authors of [8] call "cyclic effects" and the authors of [9] call "patterns of growth".

Metrics for the assessment of the law's validity: Whereas the law simply states that the evolution of software is feedback regulated, its experimental validation in the area of software systems is typically supported by the observation of a recurring pattern of smooth expansion of the system's size that is interrupted with releases with size reductions or abrupt growth. Moreover, due to a previous wording of the law (e.g., see [8]) that described change to follow a normal distribution, the experimental assessment included the validation of whether growth demonstrates oscillations around an average value [7–9]. The ripples in the size of the system are assumed to indicate the existence of feedback in the system: positive feedback results in the system's expansion and negative feedback involves perfective maintenance coming with reduced rate of growth (which is not due to functional growth but re-engineering towards better code quality) – if not with system shrinking (due to removal of unnecessary parts or their merging with other parts).

Assessment: We organize the discussion of our findings around size and growth, both of which demonstrate some patterns, although not the ones expected by the previous literature.

Size: The evolution of size can be observed in Figs. 2–4. Concerning the issue of a recurring, fundamental pattern of smooth expansion, interrupted with abrupt changes or, more generally, versions of perfective maintenance, we have to say that we simply cannot detect the behavior that Lehman did (contrast Figs. 2–4 to the respective figures of articles [7,8]): in sharp contrast to the smooth *baseline* growth that Lehman has highlighted, the evolution of the size of the studied database schemata provides a landscape with a large variety of *sequences of the following three fundamental behaviors*.

- In all schemata, we can see *periods of increase*, especially at the beginning of their lifetime or after a large drop in the schema size. This is an indication of positive feedback, i.e., the need to expand the schema to cover the information needs of the users – especially since the overall trend in almost all of the studied databases is to see an increase in the schema size as time passes.
- In all schemata, there are *versions with drops in schema size*. Those drops are typically sudden and steep and usually take place in short periods of time. Sometimes, in fact, these drops are of significantly larger size than the typical change. We can safely say that the existence of these drops in the schema size indicates perfective maintenance and thus, the existence of a negative feedback mechanism in the evolution process.
- In all schemata, there are *periods of calmness*, i.e., periods of non-modification to the logical structure of the schema. This is especially evident if one observes the heartbeat of the database, where changes are grouped to very specific moments in time.

Growth and its oscillations: Growth (i.e., the difference in the size between two subsequent versions) comes with common characteristics in all datasets. In most cases, growth is small (typically ranging within 0 and 1). As Fig. 5 demonstrates, we *have too many occurrences of zero growth*, typically iterating between small non-zero growth and zero growth. Due to perfective maintenance, we also have negative values of growth (less than the positive ones). We do not have a constant flow of versions where the schema size is continuously changing; rather, we have small spikes between one and zero. Thus, we have to state that the growth comes with *a pattern of spikes*. Due to this characteristic, *the average value is typically very close to zero* (*on the positive side*) *in all datasets*, *both for tables and attributes*. There are *few cases of large change* too; we forward the reader to Law V for a discussion and to Fig. 9 for a graphical depiction of their characteristics.

The oscillations of growth demonstrates other patterns too: it is quite frequent, especially at the attribute level, to see *sequences of oscillations of large size*: i.e., an excessive positive delta followed immediately by an excessive negative growth (see Fig. 9). We do, however, observe the oscillations between positive and negative values (remember, the average value is very close to zero), much more on the positive side, however, with several occasions of excessive negative growth (clearly demonstrating perfective maintenance).

We would like to put special emphasis to the observation that *change is small* . In terms of tables, growth is mostly bounded in small values. This is not directly obvious in the charts, because they show the ripples; however, almost all numbers are in the range of $[-2..2]$ – in fact, mostly in the range $[0..2]$. Few abrupt changes occur. In terms of attributes (Fig. 6), the numbers are higher, of course, and depend on the dataset. Typically, those values are bounded within $[-20,20]$. However, the deviations from this range are not many.

In the course of our deliberations, we have observed a pattern common in all datasets: *there is a Zipfian model in the distribution of frequencies*. Observe Fig. 7 that comes with two parts, both depicting how often a growth value appears in the attributes of Ensemble. The *x*-axis keeps the delta size and the *y*-axis the number of occurrences of this delta. In the upper part we include zeros in the counting (343 occurrences out of 528 data points) and in the lower part we exclude them (to show that the power law does not hold only for the most popular value). We observe that there is a small range of deltas, between -2 and 4 that takes up 450 changes out of the 528. This means that, despite the large outliers, change is strongly biased towards small values close to zero.

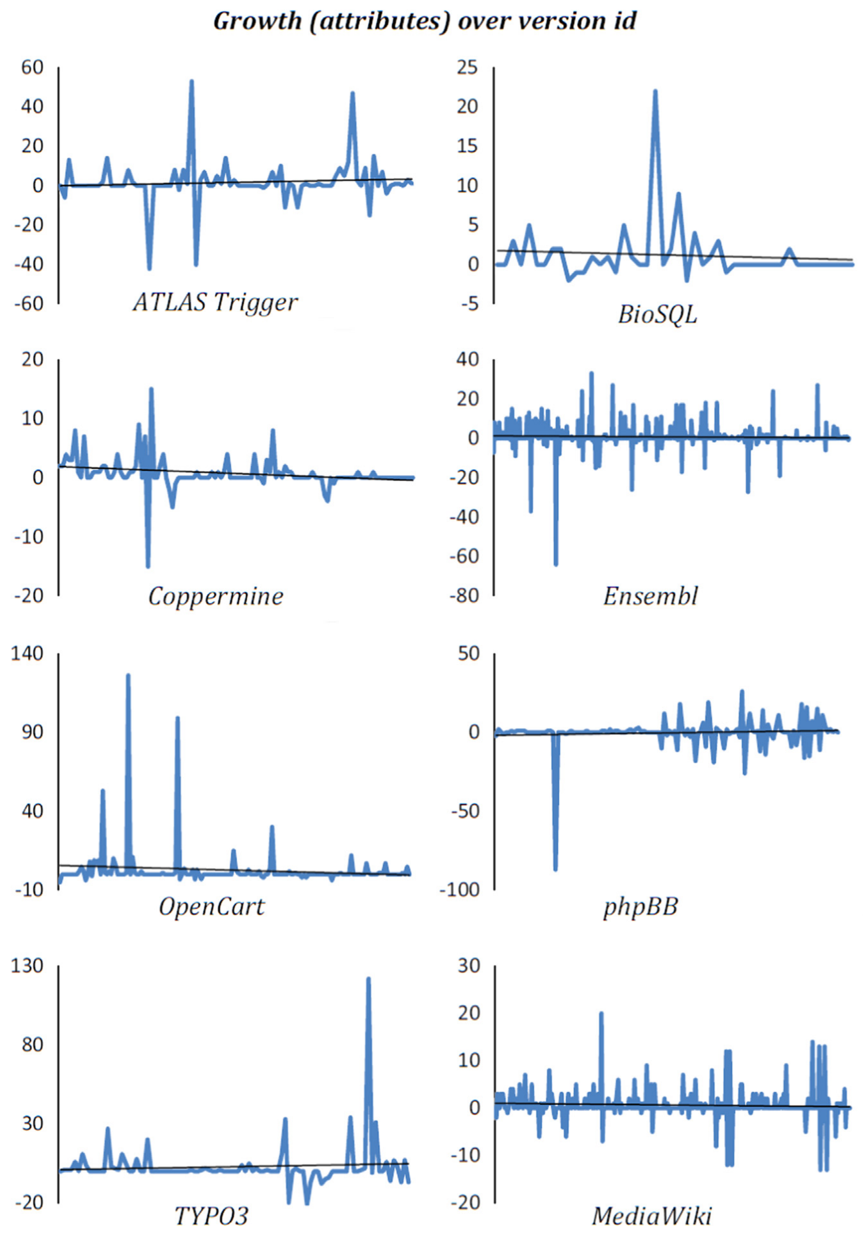


Fig. 6. Growth (attributes) over version id for all the datasets; we measure attribute growth as the difference in the total number of attributes of all tables, between two subsequent versions.

In fact, both phenomena observed here, i.e., (a) the bounded small change around zero, (b) following a Zipfian distribution of frequencies, constitute two of the patterns that are global to all datasets and without any exceptions whatsoever.

Despite the fact that change does not follow the pattern of baseline smooth growth of Lehman and the fact that change obeys a Zipfian distribution with a peak at zero, we believe that the presence of feedback in the evolution process is clear; thus the law holds.

4.1.3. *Law of Feedback System (Law VIII)*

The eighth law of software evolution is known as the law of “Feedback System”.

> E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

The main idea around this law refers to the fact that original “observation has shown that the system behaves as self-stabilizing feedback system” [14]. There is a big discussion in the literature on various components and actors whose interactions limit and guide the possible ways via which the system can evolve. We refer the interested reader to [9] for this. From our part, we do not presume to fully know the mechanics that constraint the growth of a database schema. However, we can focus to the part that there is indeed a mechanism that stabilizes the tendency for uninterrupted growth of the schema – and in fact we can try to assess

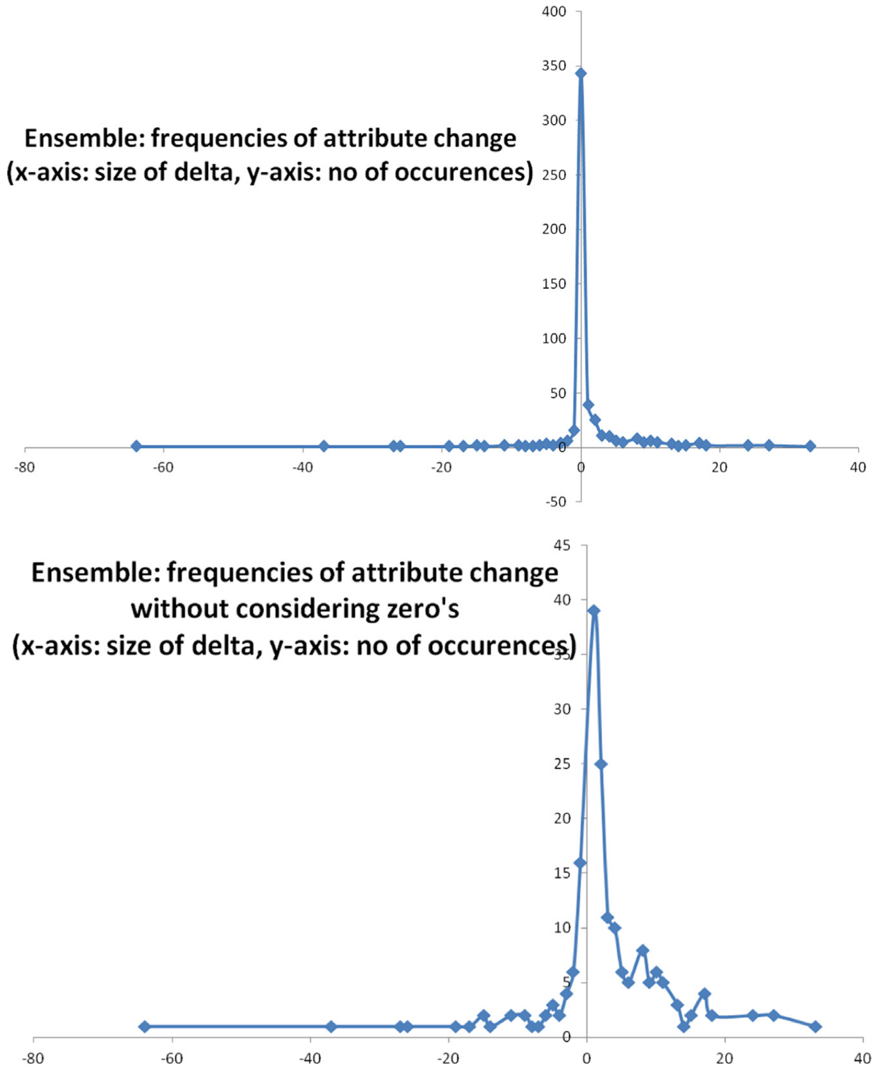


Fig. 7. Frequency of change values for Ensembl attributes.

whether this is a regressive mechanism whose behavior can be generally estimated.

Metrics for the assessment of the law's validity: To assume the law as valid we need to establish that it is possible to simulate the evolution of the schema size via an accurate formula. Following [8,15], we will perform *regression analysis* to *estimate* the number of relations for each version of the schema. We adopt the formulas found at [8,15] on the relationship of the new size of the system as a function of the previous size of it, adapted via an "*inverse square*" feedback effect. The respective formula is

$$\widehat{S}_i = \widehat{S}_{i-1} + \frac{\overline{E}}{\widehat{S}^2_{i-1}}$$
$$\overline{E} = avg(E_i), \quad i = 1 \ldots n \tag{1}$$

where $\widehat{S}$ refers to the estimated system size and $\overline{E}$ is a model parameter approximating effort. Related literature [8] suggests computing $\overline{E}$ as the average value of individual E_i, one per transition. To estimate these individual effort approximations, E_i, the authors of [8] suggest two alternative formulae:

$$E_i = (s_i - s_{i-1}) \cdot s^2_{i-1} \tag{2}$$

$$E_i = \frac{s_i - s_1}{\sum_{j=1}^{i-1} \frac{1}{s_j^2}} \tag{3}$$

Assessment: We now move on to discuss what seems to work and what not for the case of schema evolution. We will use the OpenCart dataset as a reference example; however, all datasets demonstrate exactly the same behavior.

The main challenge with formula (1) is the estimation of $\overline{E}$. As a first step, we have generalized the formulae (2) and (3) via a parameterized expression:

$$E_i = \frac{s_i - s_\alpha}{\sum_{j=\alpha}^{i-1} \frac{1}{s_j^2}} \tag{4}$$

where s_i refers to the actual size of the schema at version i and α refers to the version from which counting starts. The model of [8] comes with two values for α, specifically (i) $\alpha = i-1$ for formula (2), and (ii) $\alpha = 1$ for formula (3). The essence of the formula is that, to compute E_i, we use α previous versions to estimate effort.

Then we began our assessment. First, we assessed the formulae of [8]. In this case, we compute the average $\overline{E}$ of the individual E_i over the entire dataset. We employ four different values for α, specifically $i-1$ (last version), 1 (for the entire dataset) and 5 and 10 for the respective past versions. We depict the result in Fig. 8(top), where the actual size is represented by the blue solid line. The results indicate that the approximation modestly succeeds in predicting an overall increasing trend for all four cases, and, in fact, all four approximations targeted towards predicting an increasing tendency that the actual schema does not demonstrate. At the same time, all four approximations fail to capture the individual fluctuations within the schema lifetime.

Then, we tried to improve on this result, and instead of computing $\overline{E}$ as the total average over all the values of the dataset, we compute it as the running average (not assuming a global average, but tuning the average effort with every added release). In this case, depicted in Fig. 8 (middle), the results are less satisfactory than our first attempt.

After these attempts, we decided to alter the computation of $\overline{E}$ again. A better estimation occurred when we realized that back in 1997 people considered that the parameter $\overline{E}$ was constant over the entire lifetime of the project; however, later observations (see [9]) led to the revelation that the project was split into phases. So, for every version i, we compute $\overline{E}$ as an average over the last τ E_j values, with small values for τ (1/5/10) – contrast this to the previous two attempts where $\overline{E}$ was computed as a total average over the entire dataset (i.e., constant for all versions) or a running average from the beginning of the versions till the current one.

So, the main formula of the law is restated (and actually generalized), by replacing a global parameter $\overline{E}$ with a varying parameter $\overline{E}^i$ that can change per version (thus the superscript notation signifies the value of the effort estimation at version i). The versions used for this calculation are within the range $[\tau^s, \tau^e]$:

$$\widehat{S}_i = \widehat{S}_{i-1} + \frac{\overline{E}^i}{\widehat{S}_{i-1}^2}, \quad \overline{E}^i = avg_{j=\tau^s}^{\tau^e}(E_j) \tag{5}$$

For the three simulation attempts that we have run, we have the following configurations:

Method	Values for $[\tau^s, \tau^e]$	
Global average	τ^s: 1	τ^e: n
Running average	τ^s: 1	τ^e: $i-1$
Last τ vs. ($\tau \in \{1, 5, 10\}$)	τ^s: $i-\tau$	τ^e: $i-1$

We also decided to use the last 5 or 10 versions to compute E_i, i.e., α is 5 or 10. This has already been used in the past experiments too.

As we can see in Fig. 8(bottom), *the idea of computing the average* $\overline{E}$ *with a short memory of* 5 *or* 10 *versions produced extremely accurate results. This holds for all datasets*. This observation also suggests that, if the phases that [9] mentioned actually exist for the case of database schema, they are really small, or non-existent, and a memory of 5–10 versions is enough to produce very accurate results. The fact that this works with $\tau = 1$, and in fact, better than the other approximations is puzzling and counters the existence of phases.

We do not have a convincing theory as to why the formula works. We understand that there are no constants in the feedback system and in fact, the feedback mechanism needs a second feedback loop, with a short memory for estimating the model parameter $\overline{E}$. In plain words, this signifies that both size and effort approximation are intertwined in a multi-level feedback mechanism.

Overall, *the evolution of the database schema appears to obey the behavior of a feedback-based mechanism, as the schema size of a certain version of the database can be accurately estimated via a regressive formula that exploits the amount of changes in recent, previous versions*.

4.2. Properties of growth for schema evolution

Growth occurs as *positive feedback* to the system, in an attempt to expand the system with more functionality, or address new assumptions that make its operation acceptable, e.g., new user requirements, and an evolving operational environment. In this subsection, we study the properties of the growth.

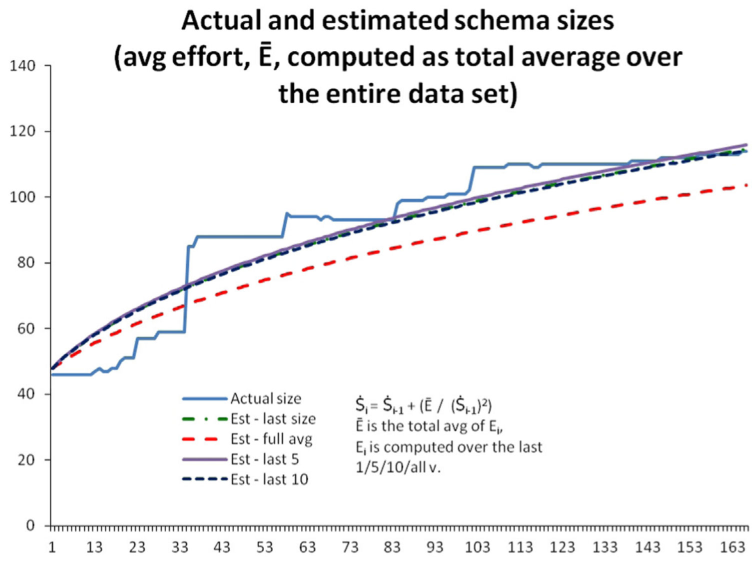
4.2.1. Law of Continuing Growth (Law VI)

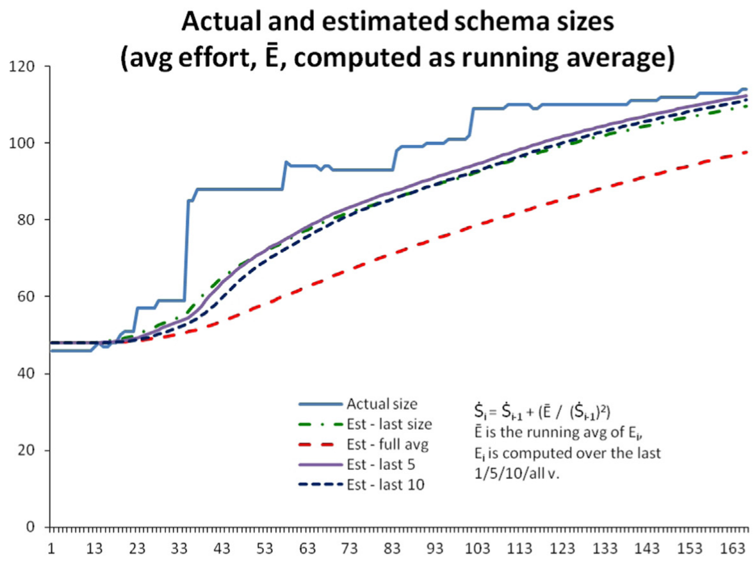
The sixth law of software evolution is known as the law of "Continuing Growth".

> The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.

The sixth law resembles the first law (continuing change) at a fist glance; however, as explained in [8], these two laws cover different phenomena. The first law refers to the necessity of a software system to adapt to a changing world. The sixth law refers to the fact that a system cannot include all the needed functionality in a single version; thus, due to non-elastic time and resource constraints, several desired functionalities of the system are excluded from a version. As time passes, these functionalities are progressively blended in the system, along with the new requirements stemming from the first law's context of an evolving world. As [9] eloquently states "the former is primarily concerned with functional and behavioral change, whereas the latter leads, in general, directly to additions to the existing system and therefore to its growth".

Metrics for the assessment of the law's validity: Possible metrics for the sixth law that come from the software engineering community [23] include LOC, number of definitions (of types, functions and global variables) and number of modules. We express again a point of concern here: it is impossible to discern, from this kind of "black-





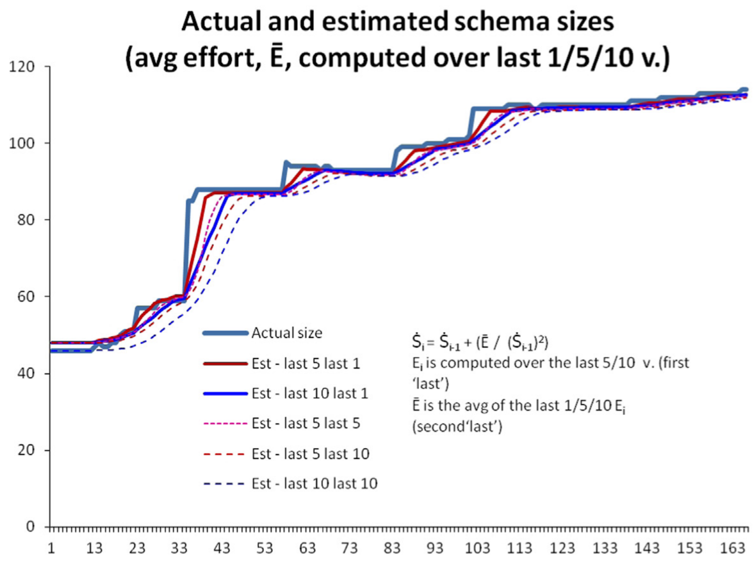


Fig. 8. Actual and estimated schema size for OpenCart via a total (top), running (middle) or bounded (bottom) averages of individual E_i. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

box" measurements, the percentage of change that pertains to the context of the law of continuing growth. Ideally, one should count the number of recorded "ToDo" functionalities blended within each version. However, we do recognize that this task is extremely hard to *automate at a large scale*. In our case, as we mainly refer to information capacity rather than physical level schema properties, we can utilize the schema size as a safe measure of observing "additions to the existing system".

Assessment: In all occasions, the schema size increases in the long run (Figs. 2–4). We frequently observe some shrinking events in the timeline of schema growth in all datasets. However, *all datasets demonstrate the tendency to grow over time*.

At the same time, we also have differences from traditional software systems: as with Law I, the term "continually" is questionable. As already mentioned (refer to Law III and Figs. 2–4), change comes with frequent (and sometimes long) periods of *calmness*, where the size of the schema does not change (or changes very little). Calmness is clearly a phenomenon not encountered in the study of traditional software systems by Lehman and acquires extra importance if one also considers that in our study we have isolated only the commits to the files with the database schema and not the commits to the entire information system that uses it: this means that there are versions of the system, for which the schema remained stable while the surrounding code changed.

Therefore we can conclude that *the law holds* (*the information capacity of the database schema is enhanced in the long run*), *albeit modified to accommodate the particularities of database schemata* (*changes are not continuous but rather, they come within large periods of calmness*).

4.2.2. *Law of Conservation of Familiarity (Law V)*

The fifth law of software evolution is known as the law of "Conservation of Familiarity".

> In general, the incremental growth (growth ratio trend) of E-type systems is constrained by the need to maintain familiarity.

As the system evolves, all the stakeholders that are associated to it (developers, users, managers, etc.) must spend effort to understand and actually, master its content and functionality. Whenever there is excessive growth in a version, the feedback mechanism tends to diminish the growth in subsequent versions, so that the change's contents are absorbed by people. Interestingly, whereas the original form of the law refers to a constant (statistically invariant) rate, the new version of the law is accompanied by explanations strongly indicating *a* "*long term decline in incremental growth and growth ratio … of all release-based systems studied*" [9]. This result came as experimental evidence from the observation of several systems, accompanied by the anecdotal evidence of a growing imbalance in volume in favor of corrective versus adaptive maintenance. Refs. [23,15] also give a corollary of the law stating that versions with high volume of changes are followed by versions performing corrective or perfective maintenance.

Metrics for the assessment of the law's validity: Ref. [9] gives a large list of possible metrics: objects, lines of code, modules, inputs and outputs, interconnections, subsystems, features, requirements, and so on. Ref. [23] proposes metrics that include (i) the growth of the system, (ii) the growth ratio of the system, and (iii) the number of changes performed in each version. We align with these tactics and use the schema growth of the involved datasets.

To validate the law we need to establish the following facts:

- The growth of the schema is not increasing over time; in fact, it is – at best – constant or, more realistically, it declines over time/version. A question, typically encountered in the literature, is: "What is the effect of age over the growth and the growth ratio of the schema?" Is it slowly declining, constant or oblivious to age? To address this question, we produce a linear interpolation of the growth per dataset to show its overall trend (Fig. 5).
- Another question of interest in the related literature is: "What happens after excessive changes? Do we observe small ripples of change, showing the absorbing of the change's impact in terms of corrective maintenance and developer acquaintance with the new version of the schema?" In this case, the pattern we can try to establish is that abrupt changes are followed by versions where developers absorb the impact of the change and produce minor modifications/corrections, thus resulting in versions with small growth following the version with significant difference in size.

Assessment: Before proceeding, we would like to remind the reader on the properties of growth, discussed in Law III of self-regulation: the changes are small, come with spike patterns between zero and non-zero deltas and the average value of growth is very close to zero (from the positive side).

Concerning the ripples after large changes, we can detect several patterns. Observe Fig. 9, depicting attribute growth for the MediaWiki dataset. Due to the fact that this involves the growth of attributes, the phenomena are amplified compared to the case of tables. Reading from right to left, we can see that there are indeed cases where a large spike is followed by small or no changes (case 1). However, within the small pool of large changes that exist overall, it is quite frequent to see sequences of large oscillations one after the other, and quite frequently being performed around zero too (case 2). In some occurrences, we see both (case 3).

Concerning the effect of age, we do not see a diminishing trend in the values of growth; however, *age results to a reduction in the density of changes and the frequency of non-zero values in the spikes. This explains the drop of the growth in almost all the studied datasets* (Fig. 5): the linear interpolation drops; however, this is not due to the decrease of the height of the spikes, but due to the decrease of their density.

The heartbeat of the systems tells a similar story: typically, change is quite more frequent in the beginning, despite the

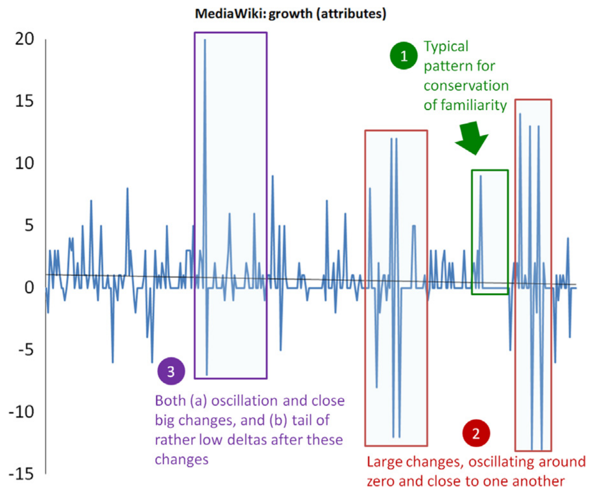


Fig. 9. Different patterns of change in attribute growth of MediaWiki (over version-id, concealed for fig. clarity).

fact that existence of large changes and dense periods of activities can occur in any period of the lifetime. Figs. 2–4 clearly demonstrate this by combining schema size and activity. This trend is typical for almost all of the studied databases. phpBB is the only exception, demonstrating increased activity in its latest versions with the schema size oscillating between 60 and 63 tables, which is actually a very small difference (as all figures are fitted to show the lines as clearly as possible, they can be deceiving as to the amount of change – phpBB is such a case).

Concerning the validity of the law, we *believe that the law is possible but not confirmed*. The law states that the growth is constrained by the need to maintain familiarity. However, the peculiarity of databases, compared to typical software systems, is that there can be other good reasons to constrain growth, such as the high degree of dependence of other modules from the database. Therefore, conservation of familiarity, although important, cannot solely justify the limited growth. The extent of the contribution of each reason is unclear.

4.2.3. *Law of Conservation of Organizational Stability (Law IV)*

The fourth law of software evolution is known as the law of "Conservation of Organizational Stability" also known as law of the "invariant work rate".

> The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime.

This is the only law with a fundamental change between the two editions of 1996 and 2006. The previous form of the law did not recognize phases in the lifetime of a project (the average effective global activity rate in an evolving E-type system is invariant over product lifetime). Plainly put, the law states that the impact of any managerial actions to improve productivity is balanced by the increasing complexity of software as time passes as well as the role of forces external to the software (availability of resources, personnel, etc.).

Metrics for the assessment of the law's validity: As [23] excellently states, it is very hard to assess effort from the data that we can typically acquire from a project, as "effort does not equate progress". Therefore, we can only approximate the work rate by observing the published versions of a system. Possible metrics [23] include (i) the number of changes per version, (ii) the average number of changes per day, and (iii) change and growth ratios.

To validate the law of conservation of organizational stability, we need to establish that the project's lifetime is divided into phases, each of which (a) demonstrates a constant growth and (b) is connected to the next phase with an abrupt change. Moreover, abrupt changes should occur from time to time and not all the time (resulting in extremely short phases).

Assessment: *If we focus on the essence of the law*, *we can safely say that it does not hold*. The heartbeats of Figs. 2–4 and the arbitrary sequencing of spikes and calmness (Figs. 5, 9) make it impossible to speak about constant growth, even in phases. The open-source nature of our cases plays a role to that too [23].

4.3. *Perfective maintenance for schema evolution*

Lehman has indicated the battle between two antagonizing processes over a fixed amount of resources for the maintenance of software [14]: on the one hand, the need to evolve the system (system growth) and on the other the "anti-regressive" effort to attack the growing complexity of the system. To achieve this, *perfective maintenance* must be performed from time to time, in order to remove redundant code, to restructure code for better maintainability and comprehension, to document the code, etc. As [9] puts it: "these activities have minor or no impact in functionality, performance or other properties of the software in execution". In this subsection, we are interested in the perfective maintenance part and we adopt the [32] definition (emphasis is ours): "*modification of a software product after delivery to provide* enhancements for users, *improvement of program documentation*, and *recoding to improve software* performance, *maintainability or other software attributes*".

4.3.1. *Law of Increasing Complexity (Law II)*

The second law of software evolution is known as the law of "Increasing Complexity".

> As an E-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.

The law states that complexity increases with age, unless effort is taken to prevent this. The rationale behind verifying the law dictates the observation of (a) an increasing trend in complexity of a software system, battled by (b) a perfective maintenance activity that attempts to reduce it and demonstrated by drops in the system size and rate of expansion.

Metrics for the assessment of the law's validity: Since we will ultimately resort to measurements for verifying the law, before proceeding further, we need to confront a fundamental problem: *the law's definition – as it stands – requires a more precise definition of complexity*. Unfortunately, complexity is a meta-property, practically involving a wide spectrum of specific measurable properties of software. To give an example, Fenton and Pfleegler [33] mention four kinds of complexity: (i) *problem complexity* (computational complexity of the underlying problem), (ii) *algorithmic complexity* (of the algorithm eventually implemented to solve the problem), (iii) *structural complexity* (typically measured as the control flow or class hierarchy or modularity structure) and (iv) *cognitive complexity* (measuring the effort required to understand the software). Lehman and Ramil [9] take a more process-oriented approach and refer to *application and functional complexity*, *specification and requirements complexity*, *architectural complexity*, *design and implementation complexity* and *structural complexity*.

Unfortunately, all the above are very hard to define and measure, especially if measurement is to be performed on evidence automatically extracted from electronic logs or version management systems. The automatic isolation of the subset of changes that pertain to perfective maintenance is an interesting and vast topic of research; for the moment, however, it appears that we will have to resort to approximations. Related literature is based on such approximations (see for example, [34]). Notably, in the latest of Lehman's series of papers, the law is supported via rationalization: the complexity increase that age brings to a system is considered responsible for the decline of the growth ratio over time (laws V and VI).

To surpass all these difficulties, we will try to assess the validity of the law based on the combination of the following observations:

First, we will focus on the essence of the law: ultimately, the law requires identifying releases or versions where perfective maintenance is performed. To actually achieve with 100% certainty would require some project management documentation that this is performed. Thus, we resort to the closest possible approximation and try to *detect versions with drops in the size and the growth of the system*. Assuming that the overall trend of the system is to grow, the existence of such points from time to time will give a strong indication of the law.

A second indication for the validity of the law is *the respect of the VIII law of feedback*, i.e., the existence of a regressive formula to which the size of the system conforms. The validity of this law would strongly insinuate the existence of a feedback-based system and therefore, the existence of negative feedback as discussed in this second law of evolution.

Third, we take a definition already found in Lehman [7,34] and *attempt to approximate the measurement of complexity as the fraction of the evolution-affected relations (i.e., the number of relations modified or added to the schema) between two subsequent versions of the schema over the difference in the number of relations of the involved versions*. This formula approximates how much effort has been invested in expanding the system over the actual difference achieved (large values demonstrate too much effort for too small change). So, for each transition, we approximate the complexity of the original schema by dividing the extent of the involved changes over the actual increment of the schema size. To understand this better, assume that we compare two transitions with the same denominator (i.e., difference in the number of relations); if one transition had more relations updated than the other, it means we paid more effort for this transition, and thus, we assume that the starting complexity is higher. More precisely, we divide the effort (number of relations that we modified in any way in a revision), by the growth (size of the result in that revision). In case the denominator is zero, we have no escape than to define complexity as zero (which is another approximation we cannot avoid).

$$complexity_i \sim \frac{relations\ handled}{|S_i - S_{i-1}|} \tag{6}$$

Assessment: Related literature typically speaks for increasing complexity [7–9,23], although there have been counter-arguments for the case of open source software [35]. In our case, *in all the datasets but phpBB, complexity, as defined in the previous paragraph, does not increase*[13] (see Fig. 10, where a linear interpolation of complexity is also depicted). The phenomenon must be coupled with the drop in change density (Law V) and although we cannot provide undisputable explanation, we offer the synergy of two causes: (a) the increasing dependence of the surrounding code to the database that makes developers more cautious to perform schema changes as they incur higher maintenance costs and (b) the success of the perfective maintenance, which results in a clean schema, requiring less corrective maintenance in the future.

Although we cannot confirm or disprove the law based on undisputed objective measurements, we have indications that the second law partially holds, albeit with completely different connotations than the ones reported by Lehman for typical software systems: in the case of database schemata, complexity, when measured as the fraction of expansion effort over actual growth, drops.

4.3.2. *Law of Declining Quality (Law VII)*

The seventh law of software evolution is known as the law of "Declining Quality".

> Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining.

The main idea behind this law concerns the fact that the software will each time be based on assumptions on the user requirements or the real world environment that will progressively be invalid. As assumptions are invalidated, action must be undertaken to maintain the affected software parts in order to reflect the actual user needs. Thus, the aging of the system, along with the increase in complexity, also calls for a reestablishment of assumptions and functionalities to serve the users' needs. Ref. [14] specifically refers to the external quality of a software system, practically expressing a system's quality as 'user satisfaction'. However, this point of view is drastically different in [9], where the viewpoint on quality is generalized to all possible kinds of quality an organization might deem necessary (based on the viewpoint of users,

[13] In our CAiSE'14 paper [31], we erroneously refer to BioSQL instead of phpBB.

managers, developers, each carrying his own interpretation and measures).

Metrics for the assessment of the law's validity: Possible metrics [23] for the internal quality of typical software systems include (i) the number of known defects associated with each version, (ii) defect density for each version, (iii) percentage of modules whose bodies have been changed. Much like the authors of [23], however, we are not really in a position to fully automate the accurate measurement of external quality as perceived by the end users, the management, etc. It is noteworthy that Lehman and Fernandez-Ramil [15,9] avoid giving any other support to the law than a logical proof: as the system expands over time, its complexity rises and thus the addressing of user requirements and removal of defects becomes more and more difficult, unless work is done to confront the phenomenon (the decline in software quality with age appears to relate to a growth in complexity that must be associated with aging).

Assessment: We follow [9] and *use logical induction to assess whether the law holds; specifically, we can assume that the law holds if it is strongly established that the laws of feedback* (*III*, *VIII*) *and complexity* (*II*) *hold*.

We have already demonstrated that the rationale behind complexity increase is not supported by our observations. At the same time, we cannot assess schema quality with undisputed means. Therefore, *we cannot confirm or disprove the law based on undisputed objective measurements*.

4.4. Threats to validity

In this subsection, we discuss threats to the validity of our conclusions. We structure our deliberations around three kinds of validity threats, specifically, construct validity, assessing the appropriateness of our measures, internal validity, assessing the possibility that cause–effect relationships are produced on an erroneous interpretation of causality, and external validity, assessing the extent to which our results can be generalized.

4.4.1. Construct validity

Construct validity concerns the appropriateness of the employed measures for the theoretical constructs they purportedly assess. In our case, to assess construct validity, we review the appropriateness of the metrics used for each law, also with a view to the metrics used in the studies of software evolution. Fig. 11 summarizes our assessment.

I. *Continuing change*: As the goal is to establish the continuity of change, the usage of the (accurately measured) heartbeat raises no concern about its appropriateness and the validity of our results.

II. *Increasing complexity*: The main metric to assess this law is the schema complexity. As we mentioned before, we do not have a way to accurately measure the complexity of a database schema as similar studies have done with software's complexity. We approximate the complexity with the effort spent between two schema versions divided by the increment in size between those versions. The later can be accurately measured but this is not the case with the effort. Effort cannot be measured from the data that we have extracted for the databases that we studied. The only accurate way to measure effort would be to have the actual man-hours that every developer has spent in the development of the database. Moreover, given the fact that databases are parts of larger software ecosystems, the possibility of accurately assessing

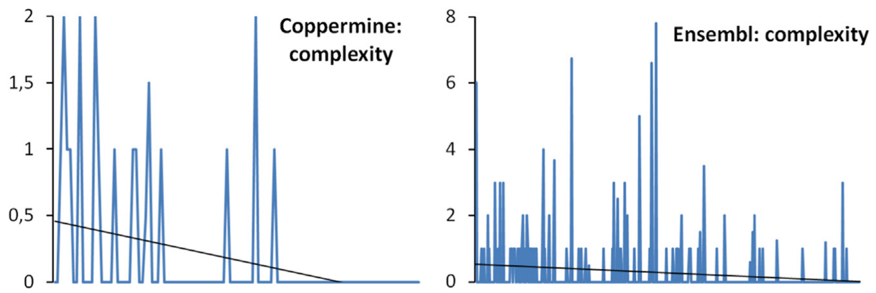


Fig. 10. Complexity for Coppermine and Ensembl (over version-id, concealed for clarity).

	Law	Measured via ...	Appropriateness
I.	Continuing Change	Heartbeat	Appropriate for detecting change events
II.	Increasing Complexity	Complexity	Approximation via change ratio
III.	Self Regulation	Size, Growth	Appropriate for finding recurring patterns
IV.	Conserv. Org. Stability	Size	Approximation of work rate
V.	Conserv. Familiarity	Growth	Appropriate for growth rate and oscillations
VI.	Continuing Growth	Size	Appropriate for seeing schema expansion
VII.	Declining quality	Rationalization	Insufficient metrics for quality
VIII.	Feedback System	Regr. Formula	Appropriate, typically used in the past

Fig. 11. Summary of measures employed per law and their appropriateness.

effort would require a measure able to differentiate the work done on the database and the work pertaining to the rest of the software system – a possibility which we dim quite slim, in fact. On the other hand, the reasoning behind the formula used makes sense and it is consistent with the related literature. Overall, the complexity, as we approximate it, poses a threat to our construct validity that we cannot ignore; to a large extent, this is also due to the abstract wording of the law. This is also the reason why we are very skeptic towards verifying the validity of the law in the case of schema evolution. Future work needs to be invested in the area for a more solid grounding of automated complexity assessment.

III. *Self regulation*: To assess this law, we used schema size and growth as measures. Both metrics can be accurately measured. The usage of the measure is consistent with the bibliography and the intuition behind the law.

IV. *Conservation of organizational stability*: The involved metric in order to assess this law is the work rate (and the existence of periods during which it remains constant). As previously mentioned, work rate cannot be easily measured, based on the available information. To this end, we primarily use schema growth as an approximation of output, and secondarily the heartbeat as an approximation of activity, both of which are accurate. Overall, we are satisfied with our choice, as it appears that this is the best possible approximation we can get from automatically extracted data; at the same time, we have to acknowledge that it is an approximation and not an undisputed measurement of the work rate.

V. *Conservation of familiarity*: The metric used for the assessment of this law is growth, which is accurately measured. On the other hand, we have no way to indisputably know the exact mechanics behind the observations; hence, despite the accuracy of the observations, the law requires further elaboration.

VI. *Continuing growth*: For this law, we employed schema size again, which is accurately extracted by our tools and fit for assessing the law.

VII. *Declining quality*: As schema quality is not clearly defined in the area of databases, the assessment of quality via metrics requires specific studies on the topic, before we are able to converge to a widely accepted solution. Rationalization about the law has typically been used in the related literature as a solution to the problem.

VIII. *Feedback system*: The main measure we used for assessing this law is the estimated size of the database schema. This measure has previously been used in the case of software evolution, again with an approximation for the measurement of effort. However, the regression formula used is consistent with its usage in the bibliography (albeit with novelty in terms of the memory of the feedback) and all the results in all datasets are surprisingly consistent. Therefore, we believe that the specific formulae used pose no threat to validity, although a better understanding of the mechanics behind the feedback mechanism have to be part of future studies.

4.4.2. *Internal validity*

Internal validity refers to the case where a conclusion on the behavior of a dependent variable is made as a cause–effect relationship with an independent variable. We are very careful to treat our observations only as such and avoid relating the observed phenomena with specific causes without supporting evidence.

Having said that, we extend the discussion, as the observant reader might be tempted to introduce a cause–effect relationship between age (as a cause) and the following phenomena: (a) dropping density of change, (b) dropping complexity, and (c) size growth in the long run. We conjecture (but cannot prove) that we could attribute the behavior of density and complexity to the existence of a confounding variable: schema quality, improving over time due to perfective maintenance and causing the observed behavior. Still, this remains to be proved with undisputed data and metrics. For size, the confounding variable is user requirements for more information capacity; although reasonable enough (in our minds, practically certain), this is also a topic to be proved indisputably by dedicated studies.

4.4.3. *External validity*

External validity refers to the possibility of generalizing the findings of a study to a broader context. Concerning the *external validity* of our study, we repeat that its context concerns *the study of the evolution of the* <u>*logical*</u> *schema of databases in* <u>*open-source*</u> *software*. We avoid generalizing our findings to databases operating in closed environments and we stress that our study has focused only on the logical structure of databases, avoiding physical properties (let alone instance-level observations).

Concerning the validity of our study within this context, we believe we have provided a safe, representative experiment. In this study, we have targeted a significant number of database schemas that serve different purposes in the real world and come with a quite broad range of time spans. Concerning the time span, the schemas collected had an adequate number of versions from rather few (40) to quite many (500+). Despite these degrees of variability, our findings are consistent in practically all of the datasets (with few exceptions that we mentioned). Thus we believe that the case of logical database schema in open source software is well represented.

On the other hand, we would be hesitant to generalize our findings in databases in closed software or outside the scope of the logical schema. Open-source software comes with a larger development community, and less control on the development effort. This is not the case for closed software, especially when dealing with mission critical components like databases. At the same time, we have not worked with the information concerning the physical schema or the extension of the studied databases and thus, we would take the opportunity to warn the reader not to generalize the results outside the scope of a schema's information capacity as expressed by the logical-level schema.

5. Discussion

In this section, we summarize fundamental *observations* and *patterns* that have been detected in our study. We intentionally avoid the term *law*, as we do not have unshakeable evidence for their explanation: apart from the *empirical grounding*, due to a very large amount of datasets that obey the same patterns (which we believe we have fairly attained), we would require an undisputed *rationalized grounding*, i.e., a clear explanation of the underlying mechanism that guides them, also established on measured, undisputed facts.

In case the reader has skipped our discussion of threats to validity, we clarify once more that the context under which our observations are made concerns *the study of the evolution of the logical schema of databases in open-source software*. In all our subsequent deliberations, we take the above context as granted and avoid repeating it for reasons of better presentation of our results.

Before proceeding, however, to our conclusions, we devote the first part of this subsection to a discussion on the validity of the problem per se.

5.1. Does the problem make sense in the first place?

We start with a *fundamental inquiry*: Is it meaningful to try assessing Lehman's laws for schema evolution in the first place? Does it make sense to try to observe evolutionary patterns in the way schemata evolve by following Lehman's method and laws?

Surely, there are fundamental differences between the general case of E-type software systems and databases in open-source systems. First, whereas software systems export *functionality* to their users, databases, on the contrary, export *information capacity*, i.e., the ability to store data and answer queries. Second, databases are not complete and independent software systems but parts of larger information systems. Is it then meaningful to pursue this research?

Again, let us revisit the fundamental lesson learned by Lehman's laws: software systems are complex, multi-level systems, involving several stakeholders, that have to evolve or face eviction; this evolution is governed by the antagonism between (a) positive feedback, pushing the system to adapt to new environments and add new functionalities according to the users' needs and (b) negative feedback, that constrains the uncontrolled growth and complexity of the system, by imposing perfective maintenance actions that result in an improved, more maintainable internal structure of the system.

Can we replace the term ‘software systems’ with ‘database’ in the above wording? We believe we can, and the fundamental reason is that the antagonism between positive and negative feedback is there too. On the one hand, a database schema has to obey the part of the positive feedback and its moderators need to adapt, tune and expand it over time (and this concerns all kinds of databases, as well as the ones involved in open-source software). This concerns both the expansion due to user requirements concerning the availability of information and the adaptation to new environments. At the same time, growth cannot be unconstrained: developers of open-source software are highly sensitive and attentive when it comes to database-related code, as changes in the database can incur both syntactic and semantic failures. Thus, it would be reasonable to expect that leaving the schema grow without any complexity control, especially in an open-source environment where developers are not organized in a strict hierarchy, can result to maintenance nightmares. The a posteriori observations verify this intuition: we do observe schema size contractions, where renamings, restructurings and removal of tables and attributes are evident in an attempt to keep schemata clean, understandable and well-structured.

Are databases, then, mini E-type systems with a life of their own? We should be clear that we do not postulate that databases can be completely isolated from the rest of their surrounding ecosystem. Still, studying schema evolution in an attempt to discover regularities and patterns is certainly worth the effort, given the high degree of dependence of the rest of the code over the database structure. With the benefit of the hindsight, we do believe that considering the laws of Lehman as a starting point for the study of schema evolution has been a legitimate and rewarding effort as it revealed both commonalities (mainly due to the same fundamental feedback mechanism) and differences (due to the specificities of the database case) with the general theory of Lehman's laws.

5.2. Major findings

In this section, we provide a critical discussion of our findings, accompanied by concise summaries, where we also annotate each of our observations with reference to the law where we have discussed it in detail. Fig. 12 further distils these findings in a single table.

5.2.1. Is the process of schema evolution behaving like a feed-back based system? (hypothesis of the feedback-based process)

We believe that we can indeed claim that schema evolution is guided by a feedback based mechanism. Positive feedback brings the need to increase the information capacity of the database, resulting in expansion of the number of relations and attributes over time. At the same time, there is negative feedback too, from the need to do some house-cleaning of the schema for redundant attributes or restructuring to enhance schema quality. We have also observed that the inverse square models for the prediction of size expansion holds for all the eight schemata that we have studied. However, we do not come with a good explanation as to why this holds. The supporting observations in this context can be listed as follows:

- As an overall trend, the information capacity of the database schema is enhanced – i.e., the size grows in the long term (VI).
- The existence of perfective maintenance is evident in almost all datasets with the existence of relation and attributes' removals, as well as observable drops in growth and size of the schema (sometimes large ones). In fact, growth frequently oscillates between positive and negative values (III).
- The schema size of a certain version of the database can be accurately estimated via a regressive formula that exploits the amount of changes in recent, previous versions (VIII).

As in all feedback-based systems, the negative feedback prevents the uncontrolled growth and retains the quality of the schema at a high level, allowing thus the subsequent

LAW & RESEARCH QUESTIONS	MEASURES	FINDINGS & COMMENTS	DATASETS
I Continuing change The schema is continually adapted	Heartbeat	The schema is adapted in the long run, albeit not continually, but in focused periods of modification	All datasets abide by the law, with two exceptions: - BioSQL, with a "sleep" of some years - phpBB with a turbulence period
III Self-regulation - Schema size expands with recurring patterns of smooth expansion, interrupted by abrupt change - Existence of shrinking versions (negative feedback)	Size	- Patterns of change include (a) expansion, (b) shrinking and (c) stability; differently from the expression of the law - Perfective maintenance is evident	All datasets without exceptions
- Change normally distributed around an average value	Growth	- Growth is small, typically close to zero, following a Zipfian model - Oscillations of large size do exist	
VIII Feedback System We can estimate the schema size via regressive formula	Regression analysis	Size estimation can be achieved; out of the different alternatives for effort estimation, the ones with small time window work better	All datasets without exceptions
VI Continuing growth The schema size is increasing in the long run	Size	Size increases in the long run, indeed, albeit not continually, but in focused periods of modification	All datasets without exceptions
V Conservation of familiarity - The average growth between versions is slowly declining	Heartbeat, Size	- The linear interpolation of growth typically drops or stays stable; importantly, the frequency of change declines	All datasets except for Atlas and BioSQL (with some extra activity in the end of their lifetimes)
- What happens after excessive changes?	Growth	- Spikes are followed by all possible combinations (calmness, other spikes, large oscillations around zero)	All datasets exhibit various patterns
IV Invariant work rate Avg. work-rate is constant within phases of smooth growth, connected with bursts of effort	(approximate) Heartbeat & Size	There is are no phases of constant growth; albeit periods of stability connected via focused periods of modifications	All datasets without exceptions
II Increasing complexity Complexity increases over time	(approximate) Schema Complexity	Complexity drops	All datasets except for phpBB (having a turbulence period in the end)
VII Declining Quality Quality declines over time	Conjecture by logical induction	Impossible to hold as valid, as complexity (albeit approximated) seems to drop	-

Fig. 12. Summary of research questions, findings and validity over the datasets.

releases to operate smoothly. This is a true sign of *stability* : the system is maintained adequately to minimize the effects of its unavoidable subsequent modifications and continue evolving smoothly.

Overall, we can state: *Schema evolution demonstrates the behavior of a stable, feedback-regulated system, as the need for expanding its information capacity to address user needs is controlled via perfective maintenance that retains quality; this antagonism restrains unordered expansion and brings stability.*

5.2.2. Hypothesis of schema size expansion (and properties of its growth)

The size of the schema expands over time, albeit with versions of perfective maintenance due to the negative feedback. As already mentioned, the inverse square model seems to work.

The growth of the database schema does not follow a pattern of smooth growth – even considering the amendment where phases of constant growth are assumed. The expansion is mainly characterized by three kinds of phases, including (i) abrupt change (positive and negative), (ii) smooth growth, and (iii) calmness (meaning large periods of no change, or very small changes). We observe that in the case of schema evolution, *the schema's growth* (i.e., its change from one version to the following) *mainly occurs with spikes oscillating between zero and non-zero values. The changes are typically small*, following a Zipfian distribution of occurrences, with high frequencies in deltas that involved small values of change, close to zero.

At the same time, in contrast to the case of software systems, *we observe a very strong inclination to avoid changes to the database schema*. Change in the database impacts surrounding code, so the change is constrained by the need to minimize this impact. So, we frequently see versions with no change to the information capacity of the schema and large time periods where the schema is still (or almost still). Bear in mind that we monitor only the subset of versions that pertain to the database schema and ignored any versions where the information system surrounding the database changed while the schema remained the same. This enforces our argument for the tendency towards stillness.

Although we do not believe conservation of familiarity to be the only cause, we see that the feedback mechanism of the evolution demonstrates *a reduction in the density of changes as the schema ages*. We also observe unexpected patterns of changes with sequences of high spikes, sometimes oscillating around zero. Such patterns require further investigation for their verification and explanation. The average growth is close to zero, and with the tendency to drop as time passes, not due to the diminishing of the (already small) deltas, whenever they occur, but mainly due to the diminishing of their density.

Concerning the *size* of the system, our supporting evidence has been already summarized via laws VI and VIII (see the previous paragraph). Concerning the *heartbeat* of the system, our supporting evidence for the above statements can be listed as follows:

- The database is not continuously adapted, but rather, alterations occur from time to time (I).
- Change does not follow patterns of constant behavior (IV).
- Age results in a reduction of the density of changes to the database schema in most cases (V).

Concerning the *growth* of the system, our supporting evidence for the above statements can be listed as follows:

- Growth is typically small in the evolution of database schemata, compared to traditional software systems (III). The distribution of occurrences of the amount of schema change follows a Zipfian distribution, with a predominant amount of zero growth in all datasets. Plainly put, there is a very large amount of versions with zero growth, both in the case of attributes and in the case of tables. The rest of the frequently occurring values are close to zero, too.
- The average value of growth is typically close to zero (although positive) (III) and drops with time, mainly due to the drop in change density (V).

5.2.3. *Hypothesis of perfective maintenance to fight complexity and user dissatisfaction*

We also believe that *there is sufficient evidence to support the claim that perfective maintenance is part of the process. This is mainly demonstrated by the drops in the schema size as well as the drops in activity rate and growth with age. In fact, growth frequently oscillates between positive and negative values* (*III*). Thus, based on simple reasoning, one can accept the wording of Lehman's laws on negative feedback, as they both state that quality (internal and external) declines unless confronted.

However, despite the adoption of the hypothesis for a feedback-based mechanism, we cannot adopt the corroborating observations of the related literature for software systems that accompany the two laws of negative feedback (II and VII). In the systems we have studied we observe that *age results in a reduction of the complexity to the database schema* (*II*), although we need to remember that the measurement of complexity is an approximation. The interpretation of the observation is that perfective maintenance seems to do a really good job and complexity drops with age (in sharp contrast to what is observed in the related literature for software systems where more and more effort is devoted to battle complexity). Also, in the case of schema evolution, activity is typically less frequent with age. Although one can attribute this to the inefficacy of the approximating measure, we anticipate that it should mainly be attributed to the truth lying in the essence of law II: "complexity increases unless work is done to reduce it". We conjecture that due to the criticality of the database layer in the overall information system, this process is done with care and achieves the reduction of complexity over time, coming hand in hand with the strong tendency towards minimum or no changes to the schema.

As for law VII, as already mentioned, we are even more hesitant to adopt it, as we are already in doubt towards internal quality and have no actual evidence as to what happens with external quality.

Overall: *although our research seems to keep the negative feedback laws in place in the case of schema evolution*, *this is done with* (a) *a degree of uncertainty and* (b) *with the strong indication of fundamental differences with E-type program evolution*. We would not be surprised if future research establishes with more certainty that the feedback mechanism for schema evolution improves the quality and complexity of a database as time passes.

5.3. *Opportunities for future work*

There are several opportunities for follow-up work. As one would normally expect, verifying the findings of this study with more datasets can further solidify our confidence to them. The extension of this work to evolution histories of proprietary databases in closed environments, over large periods of time, would be of extreme value; albeit one can only be pessimistic on the possibility of obtaining such data and being able to publish them. Novel developments in database technology allow the extension of this kind of study to non-relational data too. This includes all kinds of semi-structured data (evolution of XML data alone is a vast area of research, where the nesting of the elements provides transformations of the schema that are not present in the relational case), but also, the so-called "NoSQL" data, where structures like graphs and text evolve over time. In the latter case, the identification of patterns in the evolution of the data at the instance level is clearly a challenging topic of research.

A second large area of research concerns the identification of patterns in the correlation of the evolution of the database and the evolution of the surrounding applications. This involves both the alignment of the application code to the new schema and, as a reviewer of this paper has pointed out, possible workarounds in the code to avoid modifying the database. Even more challenging is the relationship of user requirements to database evolution. Remember that in order to be able to come up with results in long histories with many versions, automated processing of the available data is paramount. The possibility of automating the processing of tickets, bug reports and to-do lists in a way that can be correlated to the subsequent evolution of the database is a topic with a significant amount of technical challenge.

At the same time, the techniques used in this study provide opportunities for improvement. A first area of future research concerns the findings of aging and complexity (Law II). We need to establish better measures for complexity of database schemata and see how this complexity behaves over time. Similar considerations hold for estimating effort and work-rate by exploiting the available information in the software repositories as automatically as possible.

Finally, one should also recognize that the search for more patterns than the ones offered by Lehman's laws, via traditional or novel pattern detection mechanisms, is another important possibility for future work. Already, the observation of patterns of growth (Laws III and V), or patterns in the heartbeat of the evolution, are open issues worth investigating. Going further than that, identifying which tables are more liable to change in the future and how, or how the effort around schema evolution can be planned in advance by studying the available data are research questions with great value both for developers, who can tailor the code to be as loosely coupled as possible to the most unstable parts of the database, and project managers, who can estimate where change will be directed. We hope that in the context of such endeavors, the publicly available datasets of this paper (https://github.com/DAINTINESS-Group) can serve the research community.

Acknowledgment

We would like to thank the anonymous reviewers of both [31] and this paper for their useful comments.

This research has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

[1] D. Sjøberg, Quantifying schema evolution, Inf. Softw. Technol. 35 (1) (1993) 35–44.
[2] G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou, Metrics for the prediction of evolution impact in ETL ecosystems: a case study, J. Data Semant. 1 (2) (2012) 75–97.
[3] C. Curino, H.J. Moon, L. Tanca, C. Zaniolo, Schema evolution in wikipedia: toward a web information system benchmark, in: Proceedings of 10th International Conference on Enterprise Information Systems (ICEIS), 2008.
[4] D.-Y. Lin, I. Neamtiu, Collateral evolution of applications and databases, in: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE), 2009, pp. 31–40.
[5] S. Wu, I. Neamtiu, Schema evolution analysis for embedded databases, in: Proceedings of the 27th IEEE International Conference on Data Engineering Workshops (ICDEW), 2011, pp. 151–156.
[6] D. Qiu, B. Li, Z. Su, An Empirical analysis of the co-evolution of schema and code in database applications, in: Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), 2013, pp. 125–135.
[7] L.A. Belady, M.M. Lehman, A model of large program development, IBM Syst. J. 15 (3) (1976) 225–252.
[8] M.M. Lehman, J.C. Fernandez-Ramil, P. Wernick, D.E. Perry, W.M. Turski, Metrics and laws of software evolution—the nineties view, in: Proceedings of the 4th IEEE International Software Metrics Symposium (METRICS), 1997, pp. 20–34.
[9] M.M. Lehman, J.C. Fernandez-Ramil, Rules and Tools for Software Evolution Planning and Management, Software Evolution and Feedback: Theory and Practice, Wiley, Chichester, West Sussex, England, 2006.
[10] I. Herraiz, D. Rodriguez, G. Robles, J.M. Gonzalez-Barahona, The evolution of the laws of software evolution: a discussion based on a systematic literature review, ACM Comput. Surv. 46 (2) (2013) 1–28.
[11] M. Wermelinger, Y. Yu, A. Lozano, Design principles in architectural evolution: a case study, in: Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM), 2008, pp. 396–405.
[12] Z. Xing, E. Stroulia, Analyzing the evolutionary history of the logical design of object-oriented software, IEEE Trans. Softw. Eng. 31 (10) (2005) 850–868.
[13] M. Lehman, Programs life cycles and laws of software evolution, Proc. IEEE 68 (9) (1980) 1060–1076.
[14] M.M. Lehman, Laws of software evolution revisited, in: Proceedings of 5th European Workshop on Software Process Technology (EWSPT), 1996, pp. 108–124.
[15] M.M. Lehman, J.C. Fernandez-Ramil, D.E. Perry, On evidence supporting the FEAST hypothesis and the laws of software evolution, in: Proceedings of the 5th IEEE International Software Metrics Symposium (METRICS), 1998, pp. 84–88.
[16] S.S. Pirzada, A statistical examination of the evolution of the unix system (Ph.D. thesis), Imperial College, University of London, 1988.
[17] N.T. Siebel, S. Cook, M. Satpathy, D. Rodríguez, Latitudinal and Longitudinal Process Diversity, J. Softw. Maint. Res. Pract. 15 (1) (2003) 9–25.
[18] M.J. Lawrence, An examination of evolution dynamics, in: Proceedings of the 6th International Conference on Software Engineering (ICSE), 1982, pp. 188–196.
[19] M.W. Godfrey, Q. Tu, Evolution in open source software: a case study, in: Proceedings of the 16th IEEE International Conference on Software Maintenance (ICSM), 2000, pp. 131–142.
[20] M.W. Godfrey, Q. Tu, Growth, evolution, and structural change in open source software, in: Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE), 2001, pp. 103–106.
[21] G. Robles, J.J. Amor, J.M. Gonzalez-Barahona, I. Herraiz, Evolution and growth in large Libre software projects, in: Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE), 2005, pp. 165–174.
[22] S. Koch, Software evolution in open source projects: a large-scale investigation, J. Softw. Maint. Evol. 19 (6) (2007) 361–382.
[23] G. Xie, J. Chen, I. Neamtiu, Towards a better understanding of software evolution: an empirical study on open source software, in: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM), 2009, pp. 51–60.
[24] I. Herraiz, G. Robles, J.M. Gonzalez-Barahon, Comparison between SLOCs and number of files as size metrics for software evolution analysis, in: Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR), 2006, pp. 206–213.
[25] R. Vasa, Growth and change dynamics in open source software systems (Ph.D. thesis), Swinburn University of Technology, Australia, 2010.
[26] A. Israeli, D.G. Feitelson, The Linux kernel as a case study in software evolution, J. Syst. Softw. 83 (3) (2010) 485–501.
[27] C.A. Curino, H.J. Moon, C. Zaniolo, Graceful database schema evolution: the PRISM workbench, Proc. VLDB Endow. 1 (2008) 761–772.
[28] C. Curino, H.J. Moon, A. Deutsch, C. Zaniolo, Automating the database schema evolution process, VLDB J. 22 (1) (2013) 73–98.
[29] G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou, Policy-regulated management of ETL evolution, J. Data Semant. 13 (2009) 147–177.
[30] G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou, HECATAEUS: regulating schema evolution, in: Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE), 2010, pp. 1181–1184.
[31] I. Skoulis, P. Vassiliadis, A. Zarras, Open-source databases: within, outside, or beyond Lehman's laws of software evolution? in: Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE), 2014, pp. 379–393.
[32] IEEE, Guide to the Software Engineering Body of Knowledge (v. 3.0), IEEE Computer Society, 2014, available at ⟨http://www.computer.org/portal/web/swebok⟩, retrieved at 08 July 2014.
[33] N.E. Fenton, S.L. Pfleeger, Software Metrics—A Practical and Rigorous Approach, International Thomson, Boston, MA, USA, 1996.
[34] M.M. Lehman, J.F. Ramil, Software Evolution, in: STRL Annual Distinguished Lecture, De Montfort University, Leicester, 20 December 2001, available at ⟨http://www.eis.mdx.ac.uk/staffpages/mml/feast2/papers.html⟩, ⟨http://www.eis.mdx.ac.uk/staffpages/mml/feast2/papers/pdf/690c.pdf⟩, ⟨http://www.eis.mdx.ac.uk/staffpages/mml/feast2/papers/pdf/jfr103c.pdf⟩.
[35] J. Fernández-Ramil, A. Lozano, M. Wermelinger, A. Capiluppi, Empirical studies of open source evolution, in: Software Evolution, 2008, pp. 263–288.

Visual Maps for Data-Intensive Ecosystems

Efthymia Kontogiannopoulou, Petros Manousis, and Panos Vassiliadis

Univ. Ioannina, Dept. of Computer Science and Engineering, Ioannina, 45110, Hellas
{ekontogi,pmanousi,pvassil}@cs.uoi.gr

Abstract. Data-intensive ecosystems are conglomerations of one or more databases along with software applications that are built on top of them. This paper proposes a set of methods for providing visual maps of data-intensive ecosystems. We model the ecosystem as a graph, with modules (tables and queries embedded in the applications) as nodes and data provision relationships as edges. We cluster the modules of the ecosystem in order to further highlight their interdependencies and reduce visual clutter. We employ three alternative, novel, circular graph drawing methods for creating a visual map of the graph.

Keywords: Visualization, data-intensive ecosystems, clustered graphs.

1 Introduction

Developers of data-intensive ecosystems construct applications that rely on underlying databases for their proper operation, as they typically represent all the necessary information in a structured fashion in them. The symbiosis of applications and databases is not balanced, as the latter act as "dependency magnets" in these environments: databases do not depend upon other modules although being heavily depended upon, as database access is performed via queries specifically using the structure of the underlying database in their definition.

On top of having to deal with the problem of tight coupling between code and data, developers also have to address the disperse location of the code with which they work, in several parts of the code base. To quote [2] (the emphasis is ours): "Programmers spend between 60-90% of their time reading and navigating code and other data sources . . . *Programmers form working sets of one or more fragments corresponding to places of interest* . . . Perhaps as a result, *programmers may spend on average 35% of their time in IDEs actively navigating among working set fragments* . . . , since they can only easily see one or two fragments at a time."

The aforementioned two observations (code-data dependency and contextualized focus in an area of interest) have a natural consequence: developers would greatly benefit from the possibility of jointly exploring database constructs and source code that are tightly related. E.g., in the development and maintenance of a software module, the developer is interested in a specific subset of the database tables and attributes, related to the module that is constructed, modified or studied. Similarly, when working or facing the alteration of the structure

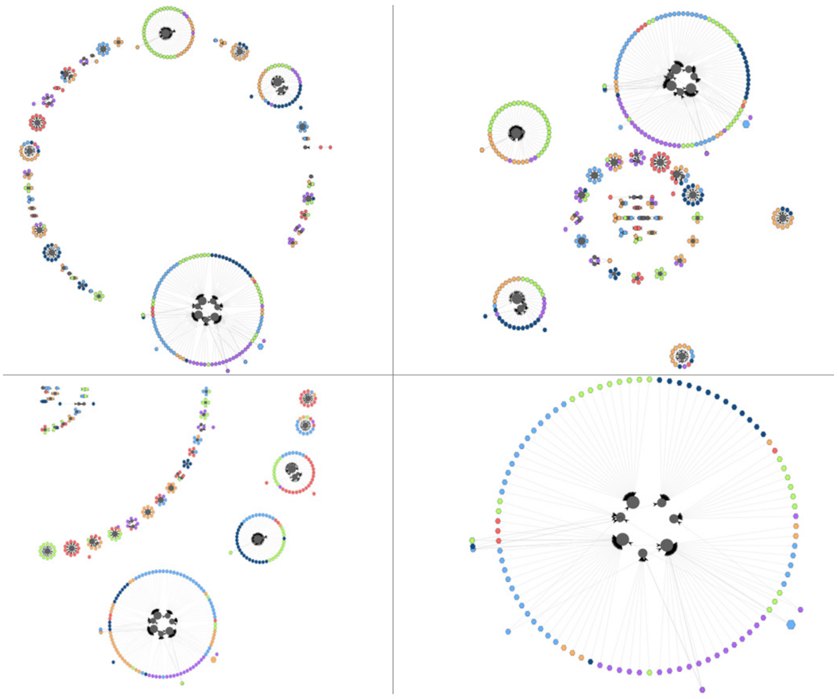


Fig. 1. Alternative visualizations for Drupal. Upper Left: Circular layout; Upper Right: Concentric circles; Lower Left: Concentric Arches. Lower Right: zoom in a cluster of Drupal.

of the database (e.g., attribute deletions or renaming, table additions, alteration of view definitions), the developer would appreciate a quick reference to the set of modules impacted by the change.

This *locality of interest* presents a clear call for the construction of a map of the system that allows developer to understand, communicate, design and maintain the code and its internal structure better. However, although (a) circular graph drawing methods have been developed for the representation general purpose graphs [11], [10], [6], and, (b) visual representations of the structure of code have been used for many decades [7], [4], [2], [3], the representation of data-intensive ecosystems has not been adequately addressed so far.

The research question that this paper addresses is the provision of a visual map of the ecosystem that highlights the correlation of the developed code to the underlying database in a way that supports the locality of interest in operations like program comprehension, impact analysis (for potential changes at the database layer), documentation etc.

Our method visualizes the ecosystem as a graph where all modules are modeled as nodes of the graph and the provision of data from a database module –e.g., a table– to a software module is denoted by an edge. To automatically detect “regions” of the graph with dense interconnections (and to visualize them accordingly) we cluster the ecosystem’s nodes. Then, we present three circular graph drawing methods for the visualization of the graph (see Fig. 1). Our first method places all clusters on a embedding “cluster” circle, our second method splits the space in layers of concentric circles and our last method employs concentric arcs. In all our methods, the internal visualization of each cluster involves the placement of relations, views and queries in concentric circles, in order to further exploit space and minimize edge crossings.

2 Graph Layout Methods for Data-Intensive Ecosystems

The fundamental modeling pillar upon which we base our approach is the *Architecture Graph* $G(V, E)$ of a data-intensive ecosystem. The Architecture Graph is a skeleton, in the form of graph, that traces the dependencies of the application code from the underlying database. In our previous research [9], we have employed a detailed representation of the queries and relations involved; in this paper, however, it is sufficient to use a summary of the architecture graph as a zoomed-out variant of the graph that comprises only of *modules* (relations, views and queries) as nodes and edges denoting data provision relationships between them. Formally, a *Graph Summary* is a directed acyclic graph $G(V, E)$ with V comprising the graph’s module nodes and E comprising relationships between pairs of data providers and consumers.

In terms of visualization methods, *the main graph layout we use is a circular layout.* Circular layouts are beneficial due to a better highlight of node similarity, along with the possibility of minimizing the clutter that is produced by line intersections. We place clusters of objects in the periphery of an embedding circle or in the periphery of several concentric circles or arches. Each cluster will again be displayed in terms of a set of concentric circles, thus producing a simple, familiar and repetitive pattern.

Our method for visualizing the ecosystem is based on the principle of *clustered graph drawing* and uses the following steps:

1. Cluster the queries, views and relations of the ecosystem, into clusters of related modules. Formally, this means that we partition the set of graph nodes V into a set of disjoint subsets, i.e., its clusters, $C_1, C_2, \ldots, C_n$.
2. Perform some initial preprocessing of the clusters to obtain a first estimation of the required space for the visualization of the ecosystem.
3. Position the clusters on a two-dimensional canvas in a way that minimizes visual clutter and highlights relationships and differences.
4. For each cluster, decide the positions of its nodes and visualize it.

2.1 Clustering of Modules

In accordance with the need to highlight locality of interest and to accomplish a successful visualization, it is often required to reduce the amount of visible elements being viewed by placing them in groups. This reduces visual clutter and improves user understanding of the graph as it applies the principle of proximity: similar nodes are placed next to each other. To this end, in our approach we use clustering to group objects with similar semantics in advance of graph drawing.

We have implemented an average-link agglomerative clustering algorithm [5] of the graph's nodes, which starts with each node being a cluster on its own and iteratively merges the most similar nodes in a new cluster until the node list is exhausted or a sued-defined similarity threshold is reached.

The distance function used in our method evaluates node similarity on the grounds of common neighbors. So, for nodes of the same type (e.g., two queries, or two tables), similarity is computed via the Jaccard formula, i.e., the fraction of the number of common neighbors over the size of the union of the neighbors of the two modules. When it comes to assessing the similarity of nodes of different types (like, e.g., a query and a relation), we must take into account whether there is an edge among them. If this is the case, the nominator is increased by 2, accounting for the two participants. Formally, the distance of two modules, i.e., nodes of the graph, M_i, M_j is expressed as:

$$dist(M_i, M_j) = 1 - \begin{cases} \dfrac{|neighbors_i \cap neighbors_j|}{|neighbors_i \cup neighbors_j|}, & if \ \nexists\ Edge(i,j) \\ \\ \dfrac{|neighbors_i \cap neighbors_j| + 2}{|neighbors_i \cup neighbors_j|}, & if\ \exists\ Edge(i,j) \end{cases} \tag{1}$$

2.2 Cluster Preprocessing

Our method requires the computation of the area that each cluster will possess in the final drawing. In our method, each cluster is constructed around three bands of concentric circles: an innermost circle for the relations, an intermediate band of circles for the views (which are stratified by definition, and can thus, be placed in strata) and the outermost band of circles for the queries that pertain to the cluster. The latter includes two circles: a circle of *relation-dedicated queries* (i.e., queries that hit a single relation) and an outer circle for the rest of the queries. This heuristic is due to the fact that in all the studied datasets, there was a vast majority of relation-dedicated queries; thus, the heuristic allows a clearer visualization of how queries access relations and views.

In order to obtain an estimation of the required space for the visualization of the ecosystem, we need to perform two computations. First, we need to determine the circles of the drawing and the nodes that they contain (this is obtained via a topological sort of the nodes and their assignment to strata, each of which is assigned to a circle), and second, we need to compute the radius for each of these circles (obtained via the formula $R_i = 3 * \log(nodes) + nodes$). Then, the outer of these circles gives us the space that this cluster needs in order to be displayed.

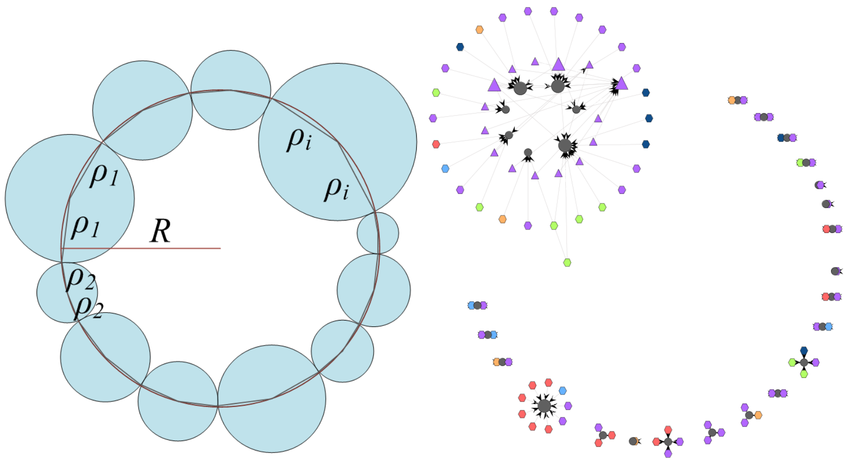


Fig. 2. Circular cluster placements (left) and the BioSQL ecosystem (right)

2.3 Layout of Cluster Circle(s)

We propose three alternative circular layouts for the deployment of the graph on a 2D canvas.

Circular Cluster Placement with Variable Angles. In this method, we use a single circle to place circular clusters on. As already mentioned, we have already calculated the radius r of each cluster. Given this input, we can also compute R, the radius of the embedding circle. We approximate the contour of the inscribed polygon of the circle, computed via the sum of twice the radius of the clusters by the perimeter of the embedding circle, which is equal to $2\pi * R$ (Fig. 2). We take special care that the layouts of the different clusters do not overlap; to this end, we introduce a white space factor w that enlarges the radius R of the cluster circle (typically, we use a fixed value of 1.8 for w). Then, $R = \sum_{i=0}^{|C|} \frac{2 * \varrho_i}{2\pi * w}$, where C is the set of clusters, and ϱ_i the radius of cluster i. As the arc around which each cluster will be placed is expanded, this leaves extra whitespace between the actually exploited parts of the clusters' arcs. Given the above inputs, we can calculate the angle ϕ that determines the sector of a given cluster, as well as its center coordinates (c_x, c_y) via the following equations:

$$\phi = 2 * \arccos\left(\frac{2 * R^2 - \varrho^2}{2 * R^2}\right),\ c_x = \cos\left(\frac{\phi}{2}\right) * R * w,\ c_y = \sin\left(\frac{\phi}{2}\right) * R * w \quad (2)$$

Concentric Cluster Placement. This method involves the placement of clusters to concentric circles. Each circle includes a different number of segments, each with a dedicated cluster. The proposed method obeys the following steps:

1. Sort clusters by ascending size in a list L^C
2. While there are clusters not placed in circles
   (a) Add a new circle and divide it in as many segments as $S = 2^k$, with k being the order of the circle (i.e., the first circle has 2^1 segments, the second 2^2 and so on)
   (b) Assign the next S fragments from the list L^C to the current circle and compute its radius according to this assignment
   (c) Add the circle to a list L of circles
3. Draw the circles from the most inward (i.e., from the circle with the least segments) to the outermost by following the list L.

Practically, the algorithm expands a set of concentric circles, split in fragments of powers of 2 (Fig. 3). As the order of the introduced circle increases, the number of fragments increases too ($S = 2^k$), with the exception of the outermost circle, where the segments are equal to the number of the remaining clusters. By assigning the clusters in an ascending order of size, we ensure that the small clusters will be placed on the inner circles, and we place bigger clusters on outer circles since bigger clusters occupy more space.

Radius Calculation. We need to guarantee that clusters do not overlap. This can be the result of two problems: (a) clusters of subsequent circles have radiuses big enough, so that they meet, or, (b) clusters on the same circle are big enough to intersect. To solve the first problem, we need to make sure that the radius of a circle is larger than the sum of (i) the radius of its previous circle, (ii) the radius of its larger cluster, and (iii) the radius of the larger cluster of the current circle. For the second problem, we compute R_i as the encompassing circle's periphery $(2 * \pi * R_i)$ that can be approximated the sum of twice the radiuses of the circle's clusters. Then, to avoid the overlapping of clusters, we set the radius of the circle to be the maximum of the two values produced by the aforementioned solutions and we use an additional whitespace factor w to enlarge it slightly (typically, we use a fixed value of 1.2 for w).

$$R_i = w * \max \begin{cases} R_{i-1} + b_{i-1} + b_i \\ \\ \frac{1}{\pi} * \sum_{j=1}^{|C|} \varrho_j \end{cases} \tag{3}$$

where (a) b_α: is the rad of biggest cluster of circle α, and (b) ϱ_j: is the rad of cluster c_j which is part of C, where C is the set of clusters of circle i.

Clusters on Concentric Arches. It is possible to layout the clusters in a set of concentric arcs, instead of concentric circles (Fig. 3). This provides better space utilization, as the small clusters are placed upper left and there is less whitespace devoted to guard against cluster intersection. Overall, this method is a combination of the previous two methods. Specifically, (a) we deploy the clusters on concentric arches of size $\frac{\pi}{2}$, to obtain a more compact layout, and (b) we partition

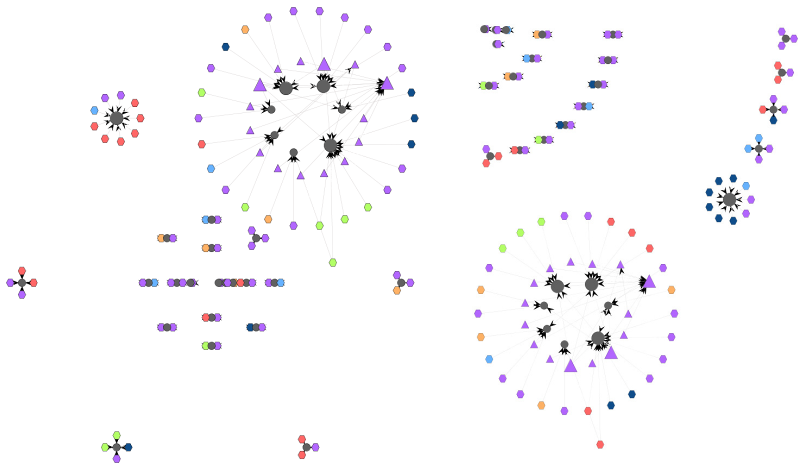


Fig. 3. Concentric cluster placement for BioSQL: circles (left), arcs (right)

each cluster in proportion to the cluster's size by applying the method expressed by equation (2).

2.4 Layout of Nodes inside a Cluster

The last part of the visualization process involves placing the internals of each cluster within the area designated to the cluster from previous computations. As already mentioned, each cluster is aligned in terms of several concentric circles: an innermost circle for relations, a set of intermediate circles for views and one or more circles for queries, as we previously stated at section 2.2. Now, since the radiuses of the circles have been computed, what remains to be resolved is the order of nodes on their corresponding circle. We order relations via a greedy algorithm that promotes the adjacency of similar relations (i.e., sharing the large amount of views and queries). Once relations have been laid out, we place the rest of the views and queries in their corresponding circle of the cluster via a traditional barycenter-based method [1] that places a node in an angle that equals the average value of the sum of the angles of the nodes it accesses.

3 To Probe Further

The long v. of our work [8] contains a full description of our method, along with its relationship to aesthetic and objective layout criteria and related experiments. Naturally, a vast area of research issues remains to be explored. First, alternative visualization methods with improved space utilization is a clear research area. Similarly, the application of the method to other types of data sets is also necessary. The relationship of graph metrics to source code properties potentially hosts interesting insights concerning code quality. Navigation guidelines (e.g., via textual or color annotation, or an annotated summary of the clusters of the graphs) also provide an important research challenge.

Acknowledgments. Prof. I. Fudos and L. Palios have made useful comments to an earlier version of this paper. This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

1. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice-Hall (1999)
2. Bragdon, A., Reiss, S.P., Zeleznik, R.C., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola Jr., J.J.: Code bubbles: rethinking the user interface paradigm of integrated development environments. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) ICSE (1), pp. 455–464. ACM (2010)
3. Caserta, P., Zendra, O.: Visualization of the static aspects of software: A survey. IEEE Trans. Vis. Comput. Graph. 17(7), 913–933 (2011)
4. DeLine, R., Venolia, G., Rowan, K.: Software development with code maps. ACM Queue 8(7), 10 (2010)
5. Dunham, M.H.: Data Mining: Introductory and Advanced Topics. Prentice-Hall (2002)
6. Halupczok, I., Schulz, A.: Pinning balloons with perfect angles and optimal area. J. Graph Algorithms Appl. 16(4), 847–870 (2012)
7. Johnson, B., Shneiderman, B.: Tree maps: A space-filling approach to the visualization of hierarchical information structures. In: IEEE Visualization, pp. 284–291 (1991)
8. Kontogiannopoulou, E.: Visualization of data-intensive information ecosystems via circular methods. Tech. rep., MT-2014-1, Univ. Ioannina, Dept. of Computer Science and Engineering (2014), `http://cs.uoi.gr/~pmanousi/publications/2014_ER/`
9. Manousis, P., Vassiliadis, P., Papastefanatos, G.: Automating the adaptation of evolving data-intensive ecosystems. In: Ng, W., Storey, V.C., Trujillo, J.C. (eds.) ER 2013. LNCS, vol. 8217, pp. 182–196. Springer, Heidelberg (2013)
10. Misue, K.: Drawing bipartite graphs as anchored maps. In: Misue, K., Sugiyama, K., Tanaka, J. (eds.) APVIS. CRPIT, vol. 60, pp. 169–177. Australian Computer Society (2006)
11. Six, J.M., Tollis, I.G.: A framework and algorithms for circular drawings of graphs. J. Discrete Algorithms 4(1), 25–50 (2006)